1970

The smiles programming system

John H. Carson Jr.
Lehigh University

THE SMILES PROGRAMMING SYSTEM

by

John H. Carson Jr.

A THESIS

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Information Science

Lehigh University

1970

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

September 15, 1970
(date)

Andrew J. Kasarda
Professor in charge

David J. Wilkman
Chairman of the Department

Table of Contents

Table of Figures

ABSTRACT

SMILES is a string manipulation language designed for the text analysis portion of the LEADER information retrieval system at Lehigh University. During the conversion of the language from the IBM 1800 computer on which it originated to a CDC 6400 computer, a formal investigation of the language was undertaken.

The results of this formal study, along with a syntax-directed language GADGET aided in the redesign of the LOADR program (this program performs the compiler and editor functions in the language). The syntax-directed technique provides a means of easy modification to the language to meet the changing demands of the LEADERMART Project.

The language is interpretive in nature, and may be considered to be executed on a virtual machine. This virtual machine, called the SMILES processor, is described from both the virtual machine standpoint and also from the program organization viewpoint. The machine has a virtual memory and utilizes paging to achieve a very large program capacity.

I. The SMILES Language, an Overview

A. History of the SMILES Language

SMILES (A String Manipulation Interpretive Language for Easy Syntax) [1] is the latest in a family of string processing languages developed at Lehigh University by the Center for Information Science for use in the text analysis portion of an information retrieval system known as LEADER [2]. The original language, called LECOM [3], was similar to COMIT [4] and was written by W. Ralph Hilton in GAP (General Assembly Program) for a GE-225 in 1966.

The next version, LECOM II [5], was written by David Reed in FORTRAN IV for an IBM 1800 computer in 1967-68. This version included expanded capabilities and was used by Reed in developing a Computational Syntatic Analyzer [6]. In the fall of 1968, it was determined that LECOM II did not possess sufficient capabilities to be useful for the application of a new text analysis program, LEGRAM [7], being developed by Michael B. Leibowitz. At this time James S. Green undertook development of LECOM III on the IBM 1800. This version became operational in June 1969, but was much more powerful than either LECOM I or LECOM II. In fact, its basic structure was quite different than either COMIT or LECOM I & II. To reflect this structure difference, the new version of LECOM was named SMILES. At that time, it was estimated that a single SMILES rule could replace approximately 100 LECOM II rules (a rule is a single statement in the language).

In June 1968, the Lehigh University Computing Center acquired a CDC 6400 computer. Its size and power were more appropriate for the development of a University-wide information retrieval network, called LEADERMART [8], being supported by the National Science Foundation (GN1055-157) under the direction of Donald J. Hillman. It was decided in June 1969 to convert the LEADER Retrieval System software from the IBM 1800 to the CDC 6400. It was during the conversion effort that this author, with the aid of James Green, took over responsibility for the conversion of the language. Conversion to the CDC 6400 began that June and was completed by September 1969. Since SMILES was written in FORTRAN IV, little difficulty was encountered in the conversion effort. The only problems encountered were conversion to different disk I/O structures and routines and the differences caused by the different internal representation of character data by the two computers.

Due to the increase in available core memory, the high-speed capabilities of the CDC 6400, and the request for additional features required for development of LEGRAM, several new features were incorporated into the language during the fall of 1969 [9]. However, these were simply extensions to the language and not changes to its basic structure.

During the development of LEGRAM, it soon became apparent that a straight-forward conversion from the IBM

1800 would not efficiently utilize the powerful capabilities of the CDC 6400, so effort was put forth by the author to correct this situation. As a result of this effort, SMILES became SMILES III (SMILES II being an intermediate stage which was never implemented). This is the version of the language now being used by the LEADERMART project at Lehigh.

B. The Nature of the Language

Although the SMILES III language retains some similarity to COMIT, there are many major differences. One of these is the notion of a Workspace, which is the key to the entire language. The Workspace is a linked list which is manipulated by a program written in SMILES III (hereafter referred to as a SMILES program).

The list is composed of "cells" which contain four data elements as shown in Figure 1.1. The first data item

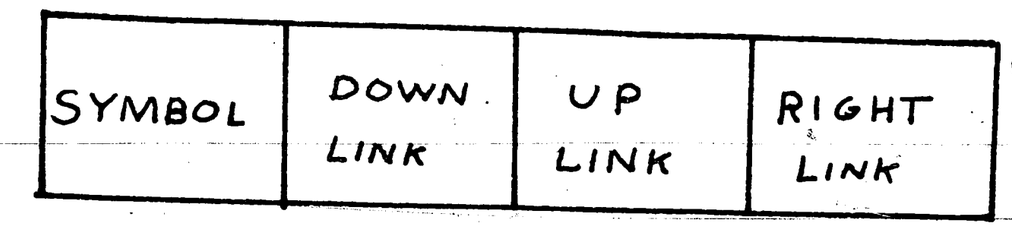


Figure 1.1. A Workspace Cell

in the cell is the symbol which identifies the cell and is the only data in the cell which is externally available to the SMILES programmer. The second data item may serve one of two purposes: It is either a pointer or linkage directive to a cell beneath it, or some external data

item which will be retrieved upon completion of the SMILES program. The third piece of data is also a pointer or linkage directive which establishes the cell which is directly above the present cell. The last data item is a pointer to the cell to the immediate right of the present cell (or it may indicate that there are no cells directly to the right).

It has been implied by definition of the pointers that the Workspace is essentially a "row-column" type of linked list. This is correct; each cell must either occur directly beneath another cell or directly to the right of a cell. To insure this, the first cell in the Workspace is always dedicated as a point of reference and all other cells occur below this cell. This reference cell is assigned the dedicated symbol, "$", and may not be modified by any SMILES program. A typical Workspace structure is shown in Figure 1.2.

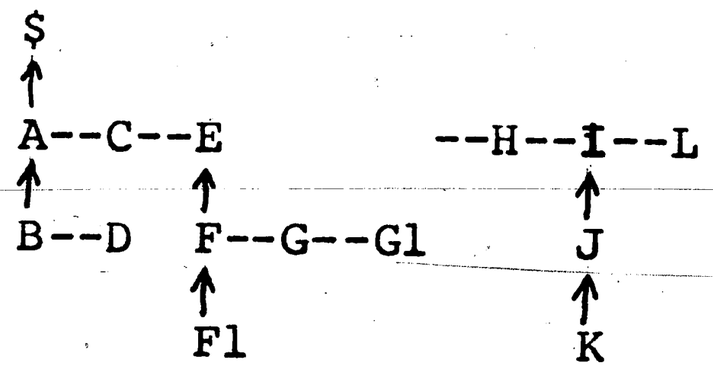


Figure 1.2. A Typical Workspace Structure

A SMILES program manipulates the Workspace structure by first declaring a syntax structure using the SMILES notation. This notation partitions the linked list into a single row string of linked sections. These sections,

called constituents, are linked only to the right between each other, with the first one starting immediately below the reference cell. Such a partitioning is shown in Figure 1.3 where the Workspace structure shown in Figure 1.2 is shown partitioned.

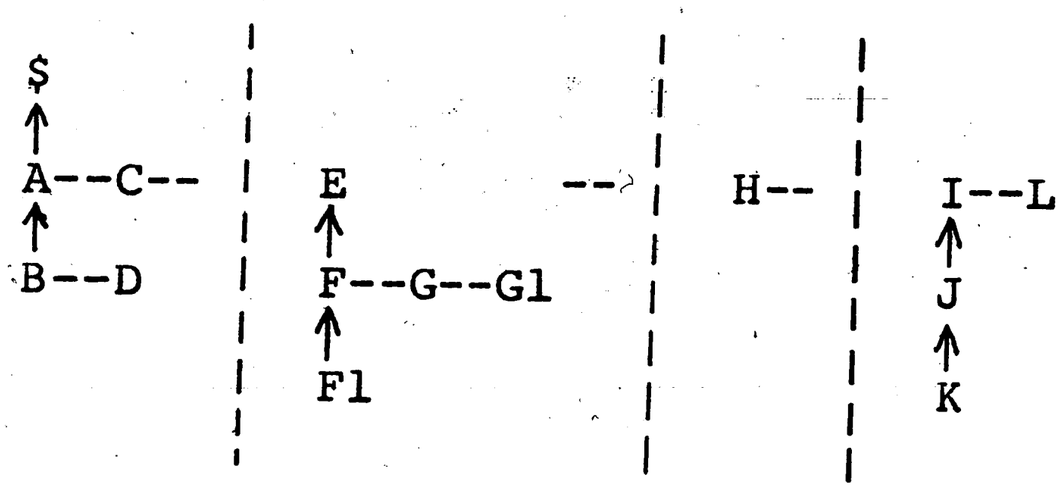


Figure 1.3. A Typical Partitioned Workspace

This string of constituents may be modified by deleting a constituent, duplicating constituents, permuting constituents, inserting single cell constituents between established constituents, or by a covering operation. This covering operation displaces the constituent to a lower level by covering it with a new symbol as shown in Figure 1.4.

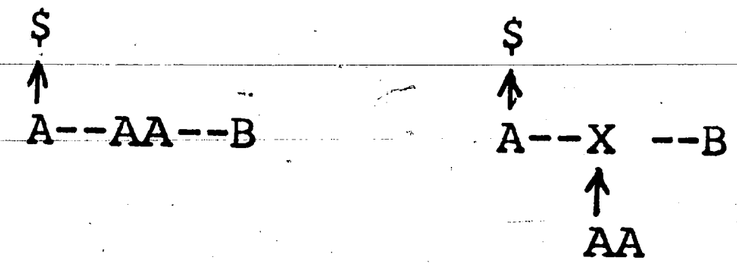


Figure 1.4. A Covering Operation on the Cell Containing "AA"

SMILES statements (or rules) are composed of four sections. As shown in Figure 1.5, the first part of a SMILES rule is the transfer label. This is used by the programmer

for reference when he wants the execution of his SMILES program to jump to that particular rule. This transfer technique will be discussed in greater detail later in this section. The next section, referred to as the "IF" section, is used to define the syntax form which the rule intends to either detect or modify. If the syntax form is recognized by the SMILES processor in the Workspace,

LABEL "*" IF THEN "*"ACCEPT , REJECT "*"

Figure 15. The Structure of a SMILES Rule

the "THEN" section is executed. The "THEN" section modifies the Workspace by referring to the constituents defined by the "IF" section. The "THEN" section may be empty (or null), in which case the rule is simply a test for the existence of a particular syntax form.

The testing ability becomes useful due to the transfer abilities of each rule. For each rule there are two transfers, an accept and a reject transfer. The transfer, the accept or the reject, that is executed depends solely on the "IF" section of the rule. If a match is made between the string defined by the syntax of the "IF"section and the Workspace, then the accept transfer is executed upon completion of the "THEN" section of the rule. If a match against the Workspace is not made, then the reject

transfer is immediately executed.

The transfer location may be either the transfer label of a rule, the name of a subroutine, or blank. If a transfer is blank, it implies that a transfer be made to the next rule in the program. If the address is the transfer label of a rule, then that rule whose label is specified is executed upon completion of the transfer operation. A subroutine CALL transfers control to a subroutine and upon returning from that subroutine, transfers control to the next rule in the calling program.

A SMILES program is composed of a main line program with up to fifty (50) subroutines. Execution of each main program or subroutine begins with the first rule and ends with the last rule in the program. When the last rule of a subroutine is executed, control returns to the main line program or whichever subroutine that called it. Subroutine calls may be recursive, either a subroutine may call itself, or a subroutine may call another subroutine which calls the original subroutine.

C. Future of the SMILES Language

Due to the interpretive nature of the language, and the structure of the LOADR (the program which acts most closely to the concept of a compiler), additions and revisions to this language are rather simple to implement. The addition of arithmetic features would be quite easy to implement, but at present, there are no demands for this type of power within the language. If such demands did

occur, another area of development and change would lie in the input (creation of the initial Workspace) and the output facilities of the SMILES execution program. Currently, two versions of SMILES exist, one streamlined version for full scale text processing which exists under LETAP (Lehigh Text Analysis Procedure) [10] under which the programmer now has control over the Workspace creation, and a more user-orientated version where the user creates the Workspace with appropriate data initialization cards.

II. The SMILES III Programming System

As mentioned in the previous chapter, SMILES programs are executed in steps. This chapter will summarize these steps so that their function will be clearly understood when they are described in greater detail later in the paper. There are three separate programs involved in the execution of a SMILES program: the LOADR program, the INCLP program, and the SMILES execution program.

The program LOADR transforms the source card images of the SMILES program into a form usable by the SMILES execution program and stores them on a disk file. The LOADR also checks for syntax errors in the source program and produces a listing of the source program and symbol table. The preparation of the source program card images involves transforming the source instructions into a condensed and slightly revised form.

Each labeled rule within the program is assigned an address within the loaded program file and a symbol table of these addresses is built in the process. Upon completion of the transformation process a check is performed to see if there were any fatal errors detected during the process and if there were, the program aborts at this point. If there were no fatal errors detected, then the address relocation process is executed and the transfer instructions are provided with the actual addresses instead of the symbolic locations. At this time, the loaded rule file is complete and ready for execution.

As the text analysis procedure LEGRAM became more in-

volved, and lengthly, it became necessary to incorporate a paging operation into the SMILES processor. The paging operation breaks the loaded rules file into segments or pages and utilizes a virtual addressing scheme [11]. In the LOADR program developed for the CDC 6400, the page boundaries for the loaded rule file were manually assigned by the SMILES programmer at suitable locations to minimize the page-in operations. It was found that this requirement on the part of the programmer often caused confusion and thereby delayed the development of LEGRAM. It was decided that this operation should be done automatically by the new LOADR program to relieve the programmer of this task and free him to develop his programs without any knowledge of this paging operation. By keeping this paging operation invisible to the SMILES programmer, it leaves the programmer free to develop his techniques without concern for program size.

The INCLP program produces the "Inclusion List" file. The Inclusion List is an ordered table of equivalence relationships between symbols referenced in the SMILES program. The programmer may declare that the symbol "AM" is included in the symbol "A" and then by referring to "A" in the program also refers to "AB". However, when "AB" is referred to in the program, "A" is not implied in any way.

The Inclusion List allows the programmer to refer to a set of symbols by including each member of the set in a general symbol. Then each time this general symbol is used,

each member of the set is also implied. This leaves the programmer free to maintain the individuality of each member of the set and still be able to refer to the set as a single symbol.

The program INCLP builds this table from data supplied by the programmer through data cards, and is stored on a disk file. This table or Inclusion List is independent of any particular SMILES program and may be used with many different loaded programs. However, the presence of the Inclusion List is not necessary for execution of a program and, if not required, may be omitted.

SMILES is an interpretive language and the execution program is an interpretive processor which executes the special SMILES instructions produced by the LOADR. Since the loaded program is kept on a disk file, it may be executed more than once without reloading the program. The SMILES execution program also initializes the Workspace from supplied data and transmits the resultant Workspace from the program execution either to a provided disk file or to the user through a listing of the Workspace.

Although these three steps are written as separate programs, they may be executed as a single process. In fact a version of SMILES was written this way for use in student projects at Lehigh. However, it is more convenient to run these programs separately since the inclusion list file and the loaded rules file are not always changed between each run.

III. The LOADR Program

A. Introduction

As stated in the previous chapter, the LOADR program behaves as the compiler for this system. The original LOADR program was written in FORTRAN IV for the IBM 1800. During the conversion to the CDC 6400, the program was entirely rewritten, again in FORTRAN IV, due to the different disk I/O structures and routines and the difference in the internal representation of character data in the two machines.

After a short period of time (about 2 months), it became apparent that this program was inadequate for use by the LEADERMART development project. Before the LOADR program was rewritten again, a study of compiler writing techniques was undertaken. Due to the experimental instability of the language processing requirements of the LEADERMART project for the development of LEGRAM, it was decided that the new LOADR program should have the following properties:

> It should have clear and meaningful diagnostic capabilities concerning syntax error detection;
>
> it should automatically assign the page boundaries for the loaded rules file to remove this burden from the programmer;
>
> and it should produce a well-formed output listing of the source program with a symbol table containing as much useful information as possible.

Upon consideration of these requirements and most heavily that of easy extensibility, it was decided that

the "Syntax Directed" technique would be the best suited for rewriting the LOADR program to meet these specifications. Major factors involved in this decision were the following:

> The general flow of a compiler written in a Syntax Directed language is much easier to design, follow, and modify;
>
> the operation performed by the LOADR program is more a problem of syntax recognition, rather than code generation;
>
> and the program could be written quicker in a higher level language and would be more easily "debugged" than if coded in assembly language.

B. Syntax-Directed Techniques in Compiling and Syntax Recognition

Two basic techniques for Syntax-Directed compiling were investigated; the Syntax-Directed technique developed by Metcalfe[12] and an application of the same basic technique implemented by Professor William Barrett at Lehigh University[13]. However, it was decided not to use either of these techniques for a common reason. Both of these languages were developed to handle algebraic languages and, as a result, use complicated techniques to handle backtracking, recursive coding, and other situations typically found in generating code for algebraic expressions. These features, in general, are not required for language processors.

As stated previously, there is little actual code generation in the loading process, for it is mostly a syntax recognition and error detection process. For this reason, it was decided to write a Syntax-Directed language

similar to Barrett's BAR-DAP, but with a simplified output scheme. The first step in the development of this Syntax-Directed language was the specification of the class of transformations and recognition problems which it would be called upon to perform. This required a study of the SMILES language syntax and a formalization of the transformations involved in the loading process.

C. The SMILES Syntax and Loading Transformations

The first task in the study of the SMILES language was to develop a formal description of the SMILES syntax. A formal backus normal description of the syntax was developed and is listed in Appendix I. The transformation processes executed by the loading operations were also formalized and appear in Appendix II. From the formalization of the transformations performed by the LOADR, it is apparent that most of the loading involves copying symbols from the input source cards to the loaded rule file. However, this process involves the elimination of blanks between the constituents of the rule; the merging of individual characters into words; and the recognition of boundaries between constituents not delimited by blanks.

However, there are some transformations which are more complicated than the copy operations, and one of the less complicated of these is shown in Figure 3.1. This transformation involves only the insertion of a "↑R" before the number and the deletion of the two periods.

There are some complex transformations in the loading

"." NUMBER "." = "↑R" NUMBER

Figure 3.1. A Simple Transformation.

process and most of these are found in the "THEN" section processor. The transformation found in Figure 3.1 was triggered by the detection of a period, but some transformations are not so easily identified. The transformation in Figure 3.2 indicates two of the more complicated transformations found in the loading process.

N_1 "/" N_2 "," N_3 = "/K" N_1 N_2 N_3

N_1 "/" N_2 = "/L" N_1 N_2

where N_i, i=1,2,3 are numbers

Figure 3.2. Two of the More Involved Transformations of the Loading Process.

It is obvious that in these types of transformations, there are no trigger characters to indicate a special case of a slash operation until the slash itself is encountered. To further complicate matters, the exact type of transformation is not known until the second number of the slash operation is processed and either the presence or absence of a comma is detected. This type of transformation can be performed by a look-ahead machine or a stack machine with a push-down list. However, by the use of some special features in the LOADR program, these complicated operations can be avoided. It was for the specifications of the transformations listed in Appendix II, that the language GADGET (Grammar Acceptance Device utilizing Generalized Enumeration Techniques) was designed, and

this language can handle these transformations with relative ease.

D. GADGET

For the most part, GADGET is a syntax-directed compiler similar to BAR-DAP with the output operations changed for convenience. To fully understand this language it is useful to visualize the operation of the language as a sequential machine [14]. Such a machine is shown in the Figure 3.3, and represents the behavior of the language.

This machine has two tapes, an input tape and an output tape. The input tape has a single head which reads a single character at a time, advancing to the right after each character has been read. However, this head may be reset to a location on the input tape by moving it to the location of the input head (this was a precaution to allow for backtracking operations). However, once moved to a location, the input pointer may never travel to the left.

The output tape has two heads and both of these can only move in a forward direction. The moving head (referred to as the M.H.) moves to the right one symbol for each symbol it writes and may never move to the left. The other head (referred to as the fixed head or F.H.) moves only upon command from the sequential machine and the only manner in which it may move is to the location of the M.H. It does not write in the conventional manner, but rewrites a single "output" on the output tape or does nothing at all. Before continuing this description of the functions

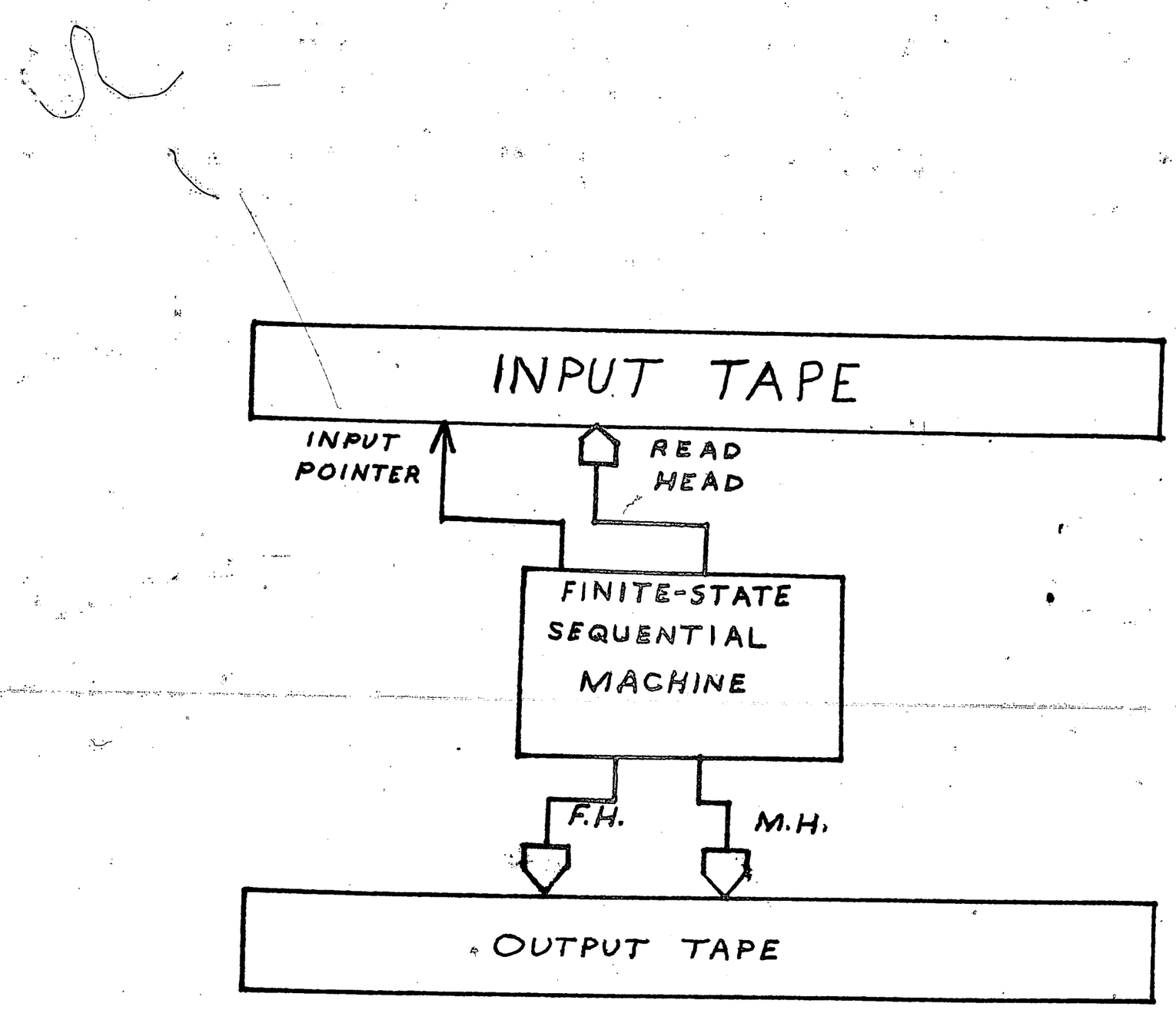


Figure 3.3. The GADGET Machine

of this machine, it seems appropriate to give a formal description of these functions.

The complete finite-state sequential machine is defined by Ginsburg [15] as a five-tuple $S = (K_s, \Sigma_s, \Delta_s, \delta_s, \lambda_s)$ where:

(i) K_s is a nonempty finite set of "states",

(ii) Σ_s is a nonempty finite set of "inputs",

(iii) Δ_s is a nonempty finite set of "outputs",

(iv) δ_s is a function (called the next-state function) which maps $K_s \times \Sigma_s$ into K_s, i.e., $\delta_s(q,I) = \bar{q}$, where q and $\bar{q}$ are in K_s and I is in Σ_s.

(v) λ_s is a function (called the "output" function) which maps $K_s \times \Sigma_s$ into Δ_s, i.e., $\lambda_s(q,I) = E$, where q is in K_s, I is in $_s$, and E is a five-tuple in Δ_s

Considering this machine in greater detail, let us describe the set of "inputs" and "outputs" referred to in the formal description and see how they relate to the loading process. The set K_s is the nonempty set of states of the machine, and each state may be considered to be the portion of the LOADR program which is executed between each reading operation.

Σ_s, the set of "inputs" to the system, is exactly the character set which is available on that machine including characters which are not legal in the SMILES language. The characters which are not used in the SMILES language must be included by the definition of the se-

quential machine. If the possibility of one of these illegal characters being used as input to the machine exists, then the machine had better be able to recognize these and perform the appropriate transformation and state transition.

The set of "outputs", Δ_s, is composed of five-tuples, in which each element of the five-tuple performs a different function. Let E be an "output" in Δ_s, E may be represented by the five-tuple $(e_1, e_2, e_3, e_4, e_5)$ where:

(i) $e_1 \in V_{op}$ (V_{op} is the set of all valid SMILES operation codes including the null symbol);

(ii) $e_2 \in V_{op}$;

(iii) $e_3 \in$ the set (R_i, ϕ), where R_i is the reset directive to the input head;

(iv) $e_4 \in$ the set (A_i, ϕ), where A_i is the advance directive to the input pointer;

(v) $e_5 \in$ the set (A_f, ϕ), where A_f is the advance directive to the fixed head.

The elements of the five-tuple are used to perform the following functions:

(i) e_1 Output to the M.H.

(ii) e_2 Output to the F.H.

(iii) e_3 Controls the reset operation of the input head.

(iv) e_4 Controls the advancing of the input tape pointer.

(v) e_5 Controls the advancing of the F.H.

When these operations are performed by the sequential ma-

chine, the only requirement pertaining to the order of execution is that e_2 must be performed before e_5.

The purpose of the fixed head is to provide a technique for handling certain non-deterministic situations which occur in the loading process. There are times when it is not known which character is to be written on the output list until several more input characters have been read. This look-ahead requirement can be accomplished by this machine without the requirement of temporary storage areas (i.e., a memory requirement). It can be seen that the transformations (i) and (ii) in the Figure 3.4 are of the form:

(i) NSNCN = S_mNNN

(ii) NSN = S NN

So until the third character in the input string is checked, it cannot be resolved whether the first character of the output string is S_m or S .

(i) N_1 "/" N_2 "," N_3 = "/M" N_1 N_2 N_3

(ii) N_1 "/" N_2 = "/L" N_1 N_2

(iii) SYM "/" N_1 = "/L" SYM N_1

(iv) SYM "/" "," = "/L" SYM **0 0**

(v) SYM "/" N_1 "," N_2 = "/M" SYM N_1 N_2

(vi) "/" N_1 = "/K" O N_1

where N_i, i=1,2,3 are numbers and SYM is any symbol.

Figure 3.4. Transformations of the "THEN" Side.

The fixed head is designed to handle transformation ambiguities of the form seen in transformations (i) and

(ii). The F.H. is stationed at the position on the output tape when the ambiguous output is written. In this case it is the initial character of the output string (either "/M" or "/L"). When the ambiguity is resolved, the correct output is written on the output tape with the fixed head. To perform all of the above operations correctly, a permutation instruction must be included. This instruction permutes the last two "outputs" on the output tape. It is realized that this permutation instruction is not in keeping with the idea of a finite-state sequential machine, but is intended to be used as a shorthand notation for an operation which would involve a tedious exhaustive construction. Since a constraint exists which limits the inputs of the machine to a finite set, then by exhaustion, a special case could be made of every output sequence which was to be permuted. To avoid this tedious task, the permutation instruction was included with the understanding that this did not remove the machine from the sequential machine class.

Before describing how the GADGET language can hangle these non-deterministic situations, the language itself will be discussed. There are six basic classes of GADGET instructions:

(i) output instructions
(ii) matching operations
(iii) pointer and head position manipulation
(iv) special utility functions
(v) symbol table manipulation instructions

The actual instructions and their explanations are listed

in Figure 3.5. Worthy of special mention are the matching operators, which attempt to match whatever is read by the input head with a specified symbol. For example, the matching operator SLASH attempts to match the next symbol from the input tape against a slash. If a slash is found, the next instruction is skipped, otherwise execution proceeds normally. When a match occurs, the read head is advanced a single symbol and if no match occurs, the read head remains at its present position.

It was decided that, in a GADGET program, the coding to handle certain syntatic cases would immediately follow the code used to detect these cases and thus if an expected syntax form is not present (the matching operation fails) there is a dedicated instruction location to contain a branch instruction to bypass this code.

To further explain the operation of the language, let us discuss the non-deterministic transformation example associated with the slash operations. In Figure 3.6, a listing is presented of the GADGET coding which will differentiate between the first two transformations shown in Figure 3.4. Starting with the first line, the NONBLK command advances the input head to the location of the next non-blank symbol on the input tape and then the ADVANCE directive moves the pointer on the input tape to the location of the read head. This combination is used many times in this example, and often will not be specifically mentioned.

OUTPUT:	
PUTIT "AA"	Write the symbol "AA" on the output list.
PUTOUT	Copy the symbol from the input tape to the output tape.
INTCHNG	Convert the last symbol on the output tape to its integer value.
CHNGIT "AB"	Write the symbol "AB" with the fixed head.
PERMUTE	Interchange the last two outputs on the output tape.
MATCHING:	
SLASH etc.	Attempt a match on the symbol from the input list and if one is found then skip the next GADGET instruction.
POINTER MANIPULATION:	
ADVANCE	Bring the input pointer to the location of the input head.
NONBLK	Move the input head to the location of the next non-blank character.
AUXILARY FUNTIONS:	
CARDIN	Read a complete SMILES rule from the input file.
PRNTSYM	Print the symbol table.
BRANCHING OPERATION:	
JMP "LOC"	Transfer execution of the program to the location names "LOC"
SYMBOL TABLE MANIPULATION:	
DEFSYM	Add the last symbol on the output list to the symbol table (by definition).
REFSYM	Add the last symbol on the output list to the symbol table (by reference).
FINDSYM	Check the symbol table to see if the last output is in the table.

Figure 3.5. The GADGET Operators.

```
                NONBLK                          01
                ADVANCE                         02
                NUMBER                          03
                JMP     NOPE                    04
                MARKIT                          05
                PUTOUT                          06
                INTCHNG                         07
                ADVANCE                         08
                SLASH                           09
                JMP     NOPE                    10
                PUTOUT                          11
                PERMUTE                         12
                NONBLK                          13
                ADVANCE                         14
                NUMBER                          15
                JMP     NOPE                    16
                PUTOUT                          17
                INTCHNG                         18
                NONBLK                          19
                ADVANCE                         20
                COMMA                           21
                JMP     CASE2                   22
*                                               23
*       THIS IS CASE 1                          24
*                                               25
                NONBLK                          26
                ADVANCE                         27
                NUMBER                          28
                JMP     NOPE                    29
                PUTOUT                          30
                INTCHG                          31
                CHNGIT /M                       32
                JMP     DONE                    33
*                                               34
*       THIS IS FOR CASE 2                      35
*                                               36
 CASE2 CHNGIT /L                                37
       JMP     DONE                             38
```

Figure 3.6 . The GADGET coding to handle transformations of class (i) and (ii)

The matching operation NUMBER checks the symbol to see if it is a number; if it is not then the string does not belong to either of the two cases in question and program execution branches to a location called NOPE (a branch to this location occurs each time the syntax of the input string fails to fit the requirements of the two transformations of the example). The fixed head is moved to the location of the M.H. by the command MARKIT and then the number is written on the output tape with the command PUTOUT. This number is in character form on the tape and is then changed into its integer value by the command INTCHNG.

Then the read head pointer is advanced to the location past the number by the ADVANCE command and a check is performed for a slash. When it is found, it is written on the output tape again with the PUTOUT command, and the number and slash are permuted with the PERMUTE operation. This leaves the fixed head ready to rewrite the slash when the ambiguity is resolved. The read head and input pointer are advanced to the next nonblank symbol and a number is checked, written out, and converted to its integer value. Then the read head and input pointer are again moved to the next nonblank character resolving the ambiguity. If this symbol is a comma (this is the case expected), the program proceeds to handle the class (i) transformation, otherwise it branches to the section labeled CASE2 to handle the class (ii) transformation.

After the comma is skipped, the read head and the input pointer are advanced to the next nonblank and the final number is checked, written out, and converted to its integer value. Then the CHNGIT command is used to rewrite the slash with the symbol "/M" and the program branches to the location DONE to indicate that it is done with this problem. If the comma were not present the program would have branched to the location CASE2 and changed the slash to the symbol "/L" before branching to the location DONE.

From the example just presented it should be clear how simple it is to write a syntax-directed translator in this language. As stated previously however, this language does not possess the capibility to handle arithmetic strings and produce the correct operator precedence relationships in the output expressions. A complete listint of the GADGET program used for the SMILES LOADR program is given in Appendix V.

E. The Symbol Table

1. Introduction

The symbol table developed by the LOADR program is a table containing all the transfer addresses, including subroutine calls, referenced in each program or subroutine. The table is used for error detection, user reference for the location of rules in the output listing, and internally by the LOADR to assign transfer addresses for execution of the program. Each entry in the table consists

of a symbol and its attribute. An attribute is essentially an indication of the usage of that particular symbol within the program.

2. Symbol Table Attributes

The attributes associated with each transfer symbol in the table indicate that symbol's usage within the program and define any errors which have occured which involve that symbol. The various attributes available in the symbol table are shown in Figure 3.7.

ATTRIBUTE	MEANING
UR	Unreferenced transfer label
UD	Undefined transfer label was referenced
SR	The transfer label is the name of a subroutine
MD	The transfer label has been defined more than once
__ (2 blanks)	The label has been both referenced and defined within the program

Figure 3.7. The Symbol Table Attributes and Their Meanings.

When each symbol is entered in the table, an attribute is assigned to the symbol. These original attributes may only be UR, UD, or SR. Upon encountering a symbol which is already in the table, it's attribute may or may not be changed depending on the conditions. A table indicating the appropriate actions to be taken under all possible conditions of this type is shown in Figure 3.8. This table points out how an attribute UD, may be changed to __ upon occurence of that symbol as a transfer address.

OCCURENCE	UR	UD	SR	__	MD
Definition	MD	__	MD	MD	MD
Reference	__	$\overline{UD}$	SR	__	MD
External declaration	$\overline{MD}$	SR	SR	$\overline{MD}$	MD

Figure 3.8. Table of Attribute Transformations by Reference of Symbols Already in the Table.

3. Entry of a Symbol into the Symbol Table

A symbol may be entered into the symbol table in one of three ways:

The symbol may be referenced and is not already in the table (handled by the routine DEFSYM);

the symbol may be used as a transfer label of a rule (handled by the routine LABEL);

or it may occur in an EXTERNAL statement defining it as a subroutine name (handled by the routine XTRNAL).

During the symbol table building operation, a counter is maintained giving the total number of symbols entered with the UD attribute minus the number of symbols with the UD attribute changed to __. This total should be zero at the completion of the loading of a single program. If the total is nonzero, this indicates that there are some symbols in the table with an UD attribute which is a fatal error.

4. Addressing in the SMILES Language

The addressing scheme used in the SMILES processor utilizes a virtual addressing technique [11]. This virtual address consists of a segment number and an offset within the segment. The segment number refers to the disk file record in the loaded rules file where the address points,

and the offset refers to a location relative to the beginning of this segment. These two components of the address are put into a single computer word as shown in Figure 3.9 to form the virtual address. Also included in this word are other data items retained for use in the output listing of the symbol table and also error message.

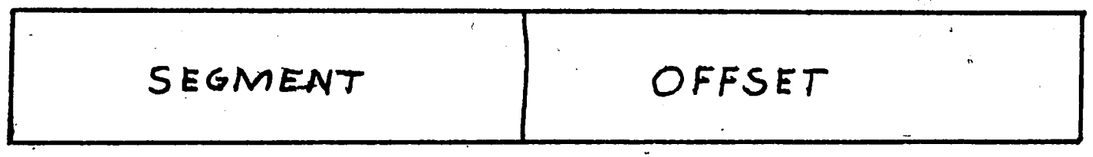


Figure 3.9. The Virtual Address

Methods of implementing this virtual address for transfer operation are discussed in Chapter VI, The SMILES Processor.

5. The Symbol Table Listing

For the benefit of the SMILES programmer, the symbol table is included in the output listing produced by the LOADR. The entries in the table are presented in alphabetical order along with their attributes and the rule number which they represent. A sample listing of the symbol table is shown in Figure 3.10.

AA	UR	563	AC	323
X4	SR	0	77	65

Figure 3.10. A sample symbol table listing

F. Implementation of the LOADR Program

The LOADR program, with the exception of a utility sort routine, is written in COMPASS (the assembly language of the CDC 6400). However, there are three levels of assembly-like coding involved. The basic level of coding is COMPASS for the utility macros which were used in writing GADGET and some of the other LOADR functions. The second level is composed of a set of macros which form the GADGET language. These are written in both COMPASS and with the utility macros previously mentioned. Finally, the LOADR program is written in GADGET, so actually all the coding is assembled by the COMPASS assembler.

In order to interface the LOADR program with the SCOPE [16] operating system of the CDC 6400, some system communication routines must be utilized. These system utilities are needed to handle disk and standard (cards and printer) I/O, overlay loading during program execution, and other file management operations. The system routine for handling the disk I/O was one of the critical factors in the design of the revised LOADR.

The disk I/O operations require both random and sequential access. The ideal characteristics of the required disk routine are:

1. No intermediate buffer operations required (this wastes time and space and also limits the number of concurrent I/O operations which may occur concurrently by the number of buffers);

2. Ability to redefine disk I/O record sizes, and

3. The ability to do concurrent disk access and processing.

The disk utility routine, INDXSEQ[17], seemed quite appropriate for the task. This routine allows the programmer to:

> issue disk read and write operations without relinquishing the central processor;
>
> allows disk record sizes to be changed during program execution with the one constraint that these records have a length which is a multiple of 64 words (this is the physical record size of the disk file and impossible to avoid unless an intermediate buffer is used for I/O blocking, and;
>
> takes up very little core memory (about 250_8 words).

It was found that by coding the LOADR in COMPASS, there was actually little need of this disk I/O routine in the LOADR program. The disk I/O operations could be done through direct calls to the system without much effort. However, INDXSEQ was written for use by FORTRAN programs such as the SMILES processor and it was intended that INDXSEQ be used exclusively in the SMILES processor. INDXSEQ requires a file header label for each file containing identification and file length parameters and is used internally by INDXSEQ. Therefore, this label had to be produced by the LOADR program and this required study into the internal operation of INDXSEQ to determine what this label looked like. Production of this header was accomplished without much trouble and the file was found to be compatable with INDXSEQ.

Worthy of mention at this time, is the routine CARDIN. While this routine does not take an active part in the transformation process of the loading operation, it is of great importance to the operation of the LOADR. This

routine is responsible for reading the SMILES source card statements from the input file. This process of preparing the source cards for processing involves: the by-passing and printing of all comment cards; processing the continuation cards to present the LOADR transformation section with a complete rule; the numbering of each complete rule for use in the symbol table and during error notification; the formulation and printing of the source card listing; and the detection of end-of-program and end-of-file marks in the source input.

Each time this routine is called, it presents a complete rule to the LOADR transformation section. This complete rule is defined by the appearance of the third asterisk after the start of the rule. Each rule, not each source card, is numbered for reference in the symbol table. When an end-of-program or end-of-file mark is detected, the CARDIN routine transfers control to the end-of-program section of the LOADR. This section performs the special operations such as symbol table sorting and printing; file management operations; and error notification.

The automatic paging requirement for the LOADR was handled in a simple manner. The page size was defined as a formal internal paramater of the LOADR program and all page operations were not to be externally visible. Each page of the loaded rule file may be comprised of only complete SMILES rules. To insure this, the two GADGET output instructions, PUTOUT and PUTIT, perform checks before placing a word on the output list to see if the cur-

rent page is full. If the page is full, then an end-of-page marker is placed where the start of the next rule would be to indicate to the SMILES processor that a page-in operation should be done before transfer to the next rule can take place. Then the page is written onto the loaded rule file and the incomplete rule is moved to the beginning of the page buffer and finally the word is written on the output list. This paging operation is actually an internal function of GADGET, and the transformation program in GADGET shows no external signs that paging is being done.

The LOADR program is formed in three overlays so that the entire LOADR program is not taking up core memory throughout the entire execution of the loading operation. This overlay structure was a natural consequence of the organization of the loading process. The loading operation is organized as a two-pass procedure with each procedure being executed only once for the entire source input. The flowchart in Figure 3.11 shows the program flow of the LOADR and also identifies the overlay partitioning of the program.

The main overlay is core resident throughout the loading operation and serves as a communication area between the other two overlays. It also performs program execution management and some bookkeeping operations. The first pass and the first primary overlay, transforms the source instructions into SMILES instructions, produces the output listing, and performs all error detecting pro-

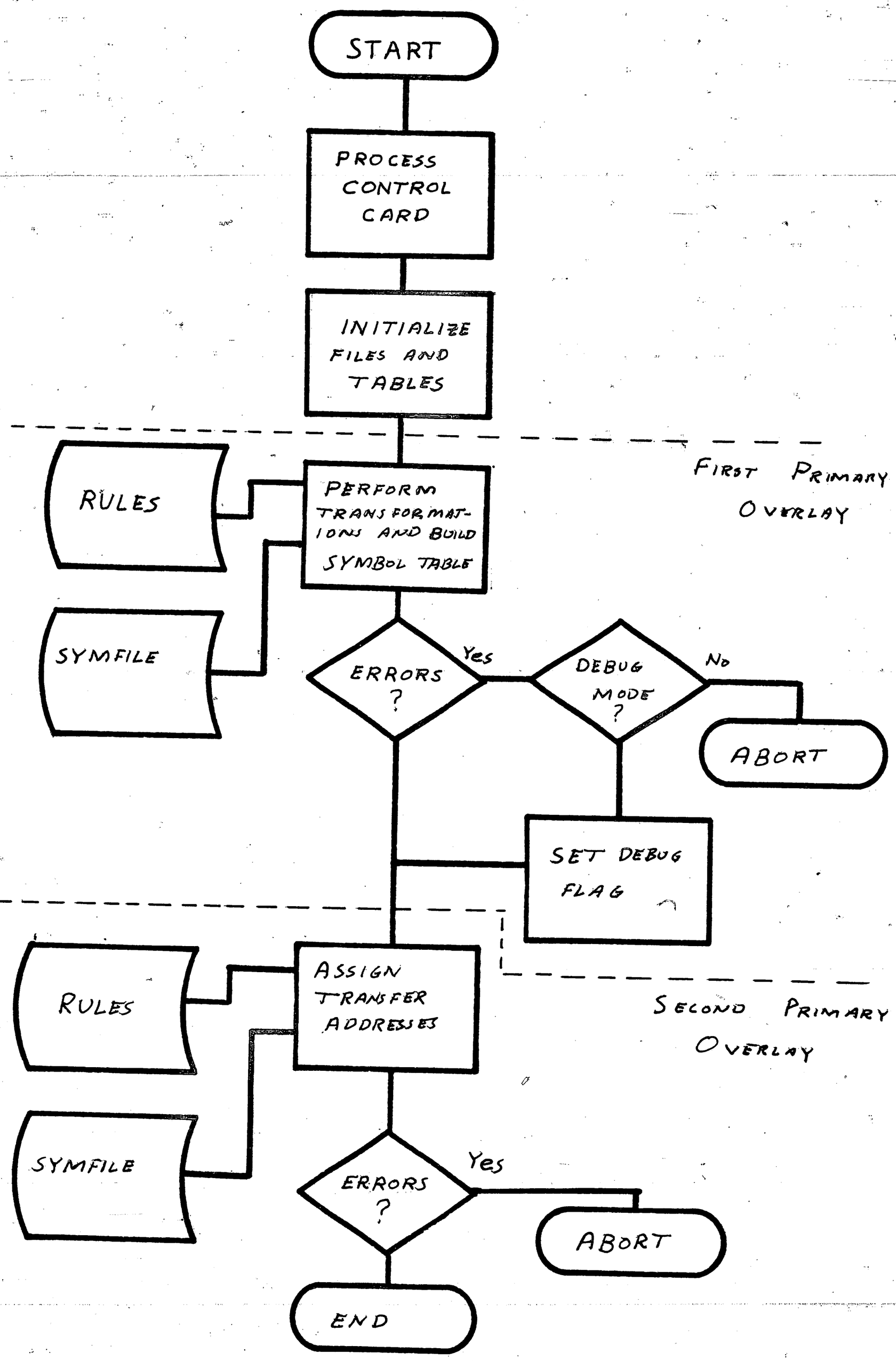


Figure 3.11. The Loading Process

cesses.

When this first pass is completed through all the programs and subroutines, the total error count is checked and only if it is zero is the second pass executed. This saves valuable computer time by not assigning address to a program which contains a fatal error and cannot be executed. This second pass, or second primary overlay, assigns the proper virtual addresses to each transfer instruction (including the subroutine addresses). The second pass is through the loaded rule file which had the space allocated for the virtual transfer addresses during the first pass. Each program or subroutine has its own symbol table containing the symbolic addresses and their respective virtual addresses for the program. Another symbol table is composed of the subroutine names and their virtual addresses.

The LOADR program takes about 25000_8 words of central memory on the CDC 6400 with over half of this being used for I/O buffers, symbol tables, and rule file storage. The LOADR can process about 4500 rules per CP (central processor) minute. A typical processing of a main program and eight subroutines containing a total of 2300 rules required 34 CP seconds, 41 PP seconds, and a total throughput time of 47 seconds. The LOADR does not require that the source input be on cards; it may use any file containing card images of up to 90 columns in length. However, only the first 72 columns of each card are considered to con-

tain useful information, but the entire card is printed. The LOADR output may also be to a file other than OUTPUT, if this is desired.

IV. INCLP, The Inclusion List Generator Program

The Inclusion list was developed as an integral part of the SMILES language. It provides the programmer with the power to specify a set of symbols by using a single symbol to identify the set. An example of an application of this technique is the problem of text analysis. As the results of a dictionary look-up procedure (the dictionary is a table consisting of text words and their category symbols) each word in the text is assigned a category symbol and these symbols are used to form the Workspace. By use of the Inclusion List feature, the programmer may specify each separate category symbol in special-case rules, but still have the ability to operate on general category classifications by using a single symbol. This allows the programmer to take advantage of both the careful categorization available by the dictionary look-up procedure and still maintain a broad classification scheme.

The program INCLP generates the Inclusion List from data supplied in card form by the programmer. This list is maintained on a disk file and consists of pairs of symbols indicating that the second symbol includes the first. For example, the pair, "AB" "AA", would indicate that "AB" is included in "AA". The program also organizes an output listing which tabulates all the symbols included in a single symbol along with a listing of the symbol pairs used as input data. These two listings allow the pro-

grammer to determine both what is included in any given symbol, and which symbols may include that symbol. A sample listing of an Inclusion List is shown in Appendix VI.

The program is written in FORTRAN IV and is very similar to the original program written for the IBM 1800 by J. S. Green. The file is organized into sets of pairs of symbols with the set identity controlled by the initial character of the first symbol in each pair. Each set is composed of those pairs of symbols whose first symbols begin with the same character. A directory to these sets is included in the file and lists the initial characters along with the locations in the list of their respective sets. Investigation shows that a binary search would produce a quicker look-up operation than this dictionary is able to give, but a binary search requires that the list be completely sorted. No sort-merge routines were written for the IBM 1800 for this purpose, but a routine is available on the CDC 6400, and this change will probably be implemented along with some other modifications to the SMILES Programming System to improve its operating efficiency.

Internally, the Inclusion List functions as an "included" list. When a look-up occurs which utilizes the Inclusion List, the Inclusion List is used to see which categories include the category from the Workspace and this including category is compared to the category from the

SMILES rule.

This restriction means that a single symbol cannot be included in more than one symbol. If the structure of the Inclusion List were changed so that when a symbol was referenced, all the categories it includes were also referenced, then the same symbol could be used as a member of many sets. This will be another topic for investigation in the future and might produce a further increase in programming power in the SMILES language.

V. The SMILES Processor

A. An Introduction

As stated earlier, the SMILES processor is an interpretive processor and was originally written by James Green in FORTRAN IV for the IBM 1800. The original version was structured in three absolute linked overlays but when the program was converted to the CDC 6400, enough core memory was available to combine these overlays into a single program. The disk I/O routines and techniques were also changed and modifications were made to compensate for differences between the two machines. The present SMILES III processor is not very different from the original converted processor, however, some new features have been added and the addressing scheme has been changed.

The SMILES processor may be considered to be a virtual computer [11] and this virtual machine is represented by the diagram in figure 5.1. It will be helpful to refer to this diagram when the internal functions of the processor are discussed in greater detail later in this chapter.

B. Execution of a SMILES Program

A SMILES program consists of one or two files; the loaded rules file and the Inclusion list file (which is not necessary for execution). Execution of a SMILES program begins by loading the SMILES processor into the core of the CDC 6400 and executioning it. From this point, execution is controlled by the SMILES processor. The first operation done by the SMILES processor is the creation of

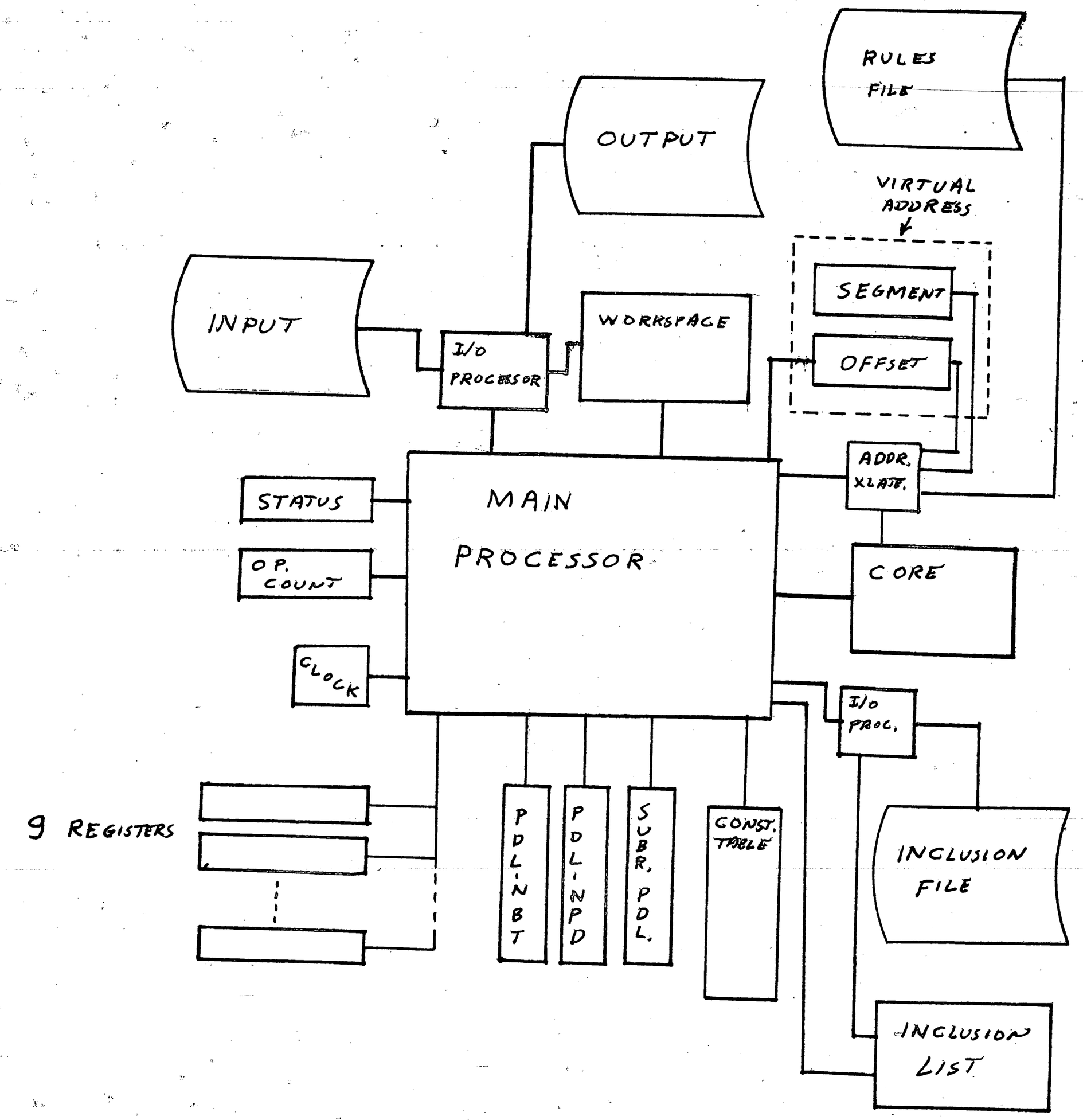


Figure 5.1. The SMILES Processor

the Inclusion list table from the Inclusion list file, if it exists. Then certain statistics tables are initialized for the run and the processor is ready to begin execution of the SMILES program. The Workspace is then initialized by the I/O routine WKSPC which reads in the symbols forming the initial string and sets up the external reference pointers in the string. Then the tables for this Workspace are initialized and the program limits are defined. These program limits are a time limit for the execution of that Workspace and a limit on the total number of rules which may be executed in the processing of the Workspace.

At this point, the first rule of the main program is executed and execution continues until normal termination of the program occurs, SMILES detected execution error occurs, the time limit runs out, the rule limit is exceeded, or the Workspace is exhausted through extensive cell creation.

The final Workspace structure is then either written onto a file for later reference, or is printed for the user. An example of a SMILES program execution is shown in Appendix IV. In either case, the time used, both CP (central processor) and PP (peripheral processor), the total number of rules considered, and the total number of Workspace cells utilized are printed to inform the user of the manner of execution performed. An exit mode is also included to identify the mode of termination, whether normal, or by occurence of an error which the code will identify.

The SMILES Processor will create a new Workspace and repeat this cycle with each set of data until the input data is exhausted.

C. Addressing and Paging in the SMILES Processor

The only addressing which occurs during execution of a SMILES program are the transfer operations. There are three possible modes of transfer and these are executed in the same manner regardless of which type of transfer, accept or reject, is being executed. All transfer operations are handled by a routine called IXFER which when given a transfer address, "A", returns only the offset within the present page in core. This function actually does much more than strip off the offset, however. First, the virtual address is checked to see if it is zero. If this is the case, then the transfer is to the next rule in the program. The offset pointer is correctly positioned and then the new value of this pointer is returned.

If the transfer is non-zero, then the segment number is examined to determine if the requested page is in core at the present time or not. If it is, then the offset is calculated and the pointer is set to the correct address. If the page requested is not in core at the present time, the address translation controller requests that this page be read in by the disk I/O routine. The page-in-core register is then set to this new page and the proper offset calculated. Then the address translation controller waits until the proper page has been read

into core and then returns the new offset to the SMILES processor. A flowchart of this process is given in Figure 5.2.

Subroutine calls are handled in much the same way except that the subroutine call is detected befor the address translation controller performs its task (a flag in the virtual address indicates a subroutine call. See Figure 3.9. Before each address is passed to the address translation controller, it is examined to see if it is a subroutine call. If a subroutine call is detected, then the return address is added to the push-down list of return addresses and the subroutine flag is turned on. (The return address is also a virtual address.) The address of the subroutine is passed to the address translation controller just as a normal transfer operation.

Subroutine execution begins with the first rule in the subroutine and continues until the last rule in the subroutine is executed. Then a flag is encountered indicating the end of a program or subroutine. When this flag is encountered, the subroutine flag is checked to see if it was a subroutine being executed. If the flag is on, indicating a subroutine termination, the last address entered into the return address push-down list is retrieved and the push-down list is examined to see if there are any return addresses remaining in it. If there are, then the subroutine flag is left on, otherwise it is turned off. Then the retrieved address is passed to the

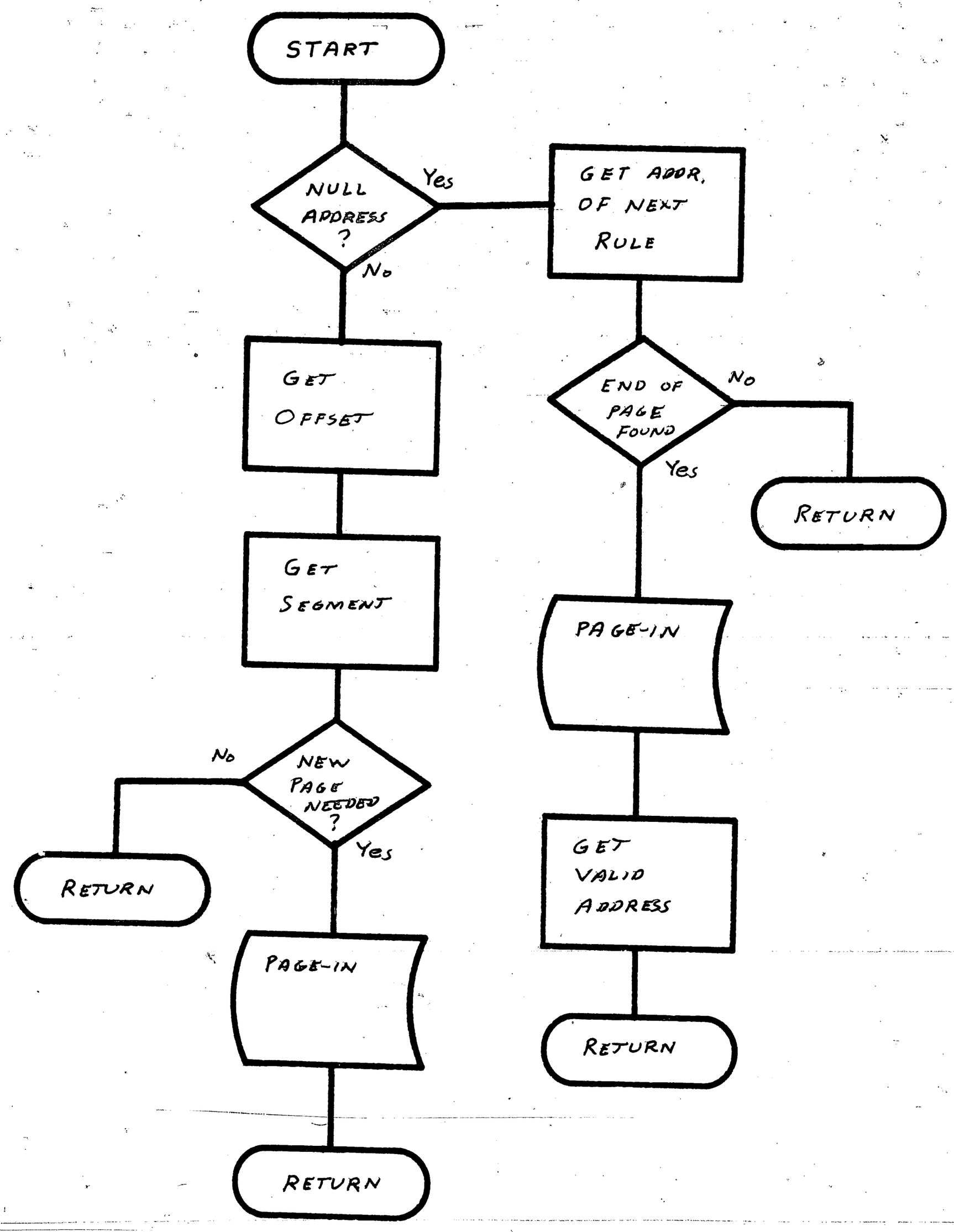


Figure 5.2. IXFER- The Virtual Address Processor

address translation controller and in this manner control is returned to the calling program.

In the virtual machine representation, the virtual address is shown composed of the location counter (the offset) and the segment counter (the required page). The page in core is kept in the present page register and the function IXFER is represented by the address translation controller.

D. Execution of a SMILES Rule

Each SMILES rule is composed of three sections, the "IF" section, the "THEN" section, and the transfer section. Execution begins with the "IF" section and in the virtual machine representation, the status or mode register is set to indicate this. Each word in the "IF" section is examined and the proper operations for this instruction are executed. There are both multiple and single word instructions, with the first words of the multiple word instructions indicating which class of a general instruction is desired. The execution of the SMILES processor is shown in the flowchart in Appendix III.

As mentioned before, each symbol from the loaded rules file is examined to determine the class of operations it indicates by searching a list of SMILES operations. Insertion of a new feature into the language is relatively easy, with just the addition of a new symbol into the operation list and a new processing section to handle the operation.

All attempts to match symbols from the SMILES program to symbols in the Workspace are done by a single routine in the SMILES processor, KONSTLK. This routine handles matching checks between the Workspace and symbols in the source program, symbols in registers, and previously defined constituents. Each comparison is checked through the Inclusion list to see if the symbol from the source program includes the category from the Workspace.

As the "IF" section of the rule attempts to find a match in the Workspace for its syntax description, it may be necessary to backtrack or restart the search operation. The rule and Workspace shown in Figure 5.3 are a typical example of this type of situation. The rule is attempting to find the string A C D E F or the string B C D E F anywhere in the Workspace. On this first pass it finds the string A C but not the D E F, at this point it must backtrack along both the Workspace and the "IF" section to correctly perform the matching operation. Then the string B C D E F is correctly identified. This backtracking is done with the aid of two push-down lists labeled NBT and NPD in the virtual machine illustration. These lists keep track of locations where backtracking could occur and allow for nested syntax definitions in the "IF" section.

During the execution of the "IF" section, the Constituent table is built for the Workspace. This table con-

sists of the starting cell number and the final cell number for each substring of the Workspace which is defined by the "IF" section syntax. These constituents form a single level string which is linked only to the left and to the right. The initial constituent is only linked to the top and to the right and the final one is only linked to the left.

Upon successful completion of the "IF" section of a rule, control is transferred to the "THEN" section of that rule. The "THEN" section operates on the Workspace through the constituents formed by the "IF" section of the rule. If the "THEN" section is empty, no action is taken on the Workspace and control is transferred to the Accept transfer section. If the "THEN" side of the rule is not empty, then the Workspace will probably be modified by the rule. With the non-empty "THEN" section, the Workspace is completely rebuilt. The reference cell containing the "$" is retained, but only those constituents which are mentioned in the "THEN" section will remain in the Workspace.

The SMILES processor scans the "THEN" section from left to right and adds constituents; modifies constituents; or adds new symbols to the Workspace as it encounters the instructions in the "THEN" section. To modify and add new cells to the Workspace, a push-down list of all the unused cells is maintained. When a cell is discarded from the Workspace, it is added to this push-down list. To add a new cell to the Workspace, this list is consulted to obtain the number of an unused cell. If

the list is exhausted, the smiles processor terminates the execution of the program for that particular Workspace and returns an error code to indicate the cause of the abnormal termination.

During this scan of the "THEN" section, each character of the "THEN" section is checked to see if it is an asterisk. This third asterisk of the rule is an indication to the SMILES processor that the "THEN" section is completed. If the "IF" section failed to match the Workspace, then the Reject transfer is executed, otherwise the Accept transfer is executed and the processing of the rule is complete.

E. Operation of SMILES in Development Mode

As stated in previous chapters, SMILES was developed to be executed under the control of LETAP. However, for developmental purposes, it was found necessary to develop a version to aid in the development of LEGRAM through the utilization of debugging or tracing aids which were not compatable with LETAP.

The developmental version builds the Workspace from data supplied on cards. This eliminates the necessity for utilizing the dictionary look-up techniques or for that matter the necessity of having a dictionary of syntatic categories. The results of the SMILES processing of the strings is returned to the programmer through a linked list drawing routine developed by Green. This routine prints the Workspace with each cell in the position determined by its pointers.

In order to locate errors in the program logic or to observe the behavior of particular sections of the program, several tracing options were developed. Originally on the IBM 1800, these options corresponded to the sense switches on the control panel, but on the CDC 6400 they correspond to internal flags in the SMILES processor. Each switch controls the status of a single tracing option and these options are:

Switch	Functions
(2)	identifies the rules whose "IF" section found a match in the Workspace.
(3)	prints the Workspace which the "IF" section of a rule accepted before the "THEN" section processes it.
(4)	identifies each rule in which the "IF" section failed to match the Workspace.
(5)	prints the Workspace by a method different than the linked method used by Green (developed for cases where the physical limitations of the printer paper prohibit proper printout of the Workspace.
(6)	prints the Workspace after the processing by the "THEN" section has been completed.

These switches are all independant, although some have no meaning unless used in conjunction with certain others. For example, the trace produces by switch 3 will have little meaning unless the rules which produced by it are identified by the trace produced by switch 2.

As the text analysis procedure became more detailed and lengthly, it became increasingly inconvenient to run

an analysis of a string with a trace in effect for the entire processing. To make tracing more convenient, a delayed turn on option was developed. This allowed the SMILES programmer to turn on individual switches after a specified number of rules had been executed. For example, he could turn on switch 2 after 250 rules had been executed, and then switch 3 after 300 rules had been executed.

This option helped eliminate the problem, but better still was the option to turn on the switches over a specified interval. This allowed the programmer to turn switches on for intervals of execution and for as many intervals as he wished. For example, switch 3 may be turned on for the intervals 250 to 300 and 450 to 600. This ability to be able to turn off a switch makes it possible to trace program execution through an interval and still see the final result without all the intermediate operations being disclosed.

It has turned out that these tracing features proved to be invaluable in the development of LEGRAM. It appears that they are also valuable tools for the study of the SMILES operations.

F. Operations under LETAP

LETAP (Lehigh Text Analysis Procedure) is the monitor program for the processing operations of the LEADERMART system. The SMILES processor operates under it in production mode in a slightly different manner than it does in program development mode. SMILES is a single overlay

under LETAP and operates as a two-tape machine. The input tape is produced by LETAP consisting of strings of category assignments for the words in the text to be analyzed and the output is a string of symbols indicating the sentential structure of the sentence. This is used to locate and identify the different types of noun phrases within each sentence.

Each category assignment in the input tape carries along with it an external word number which uniquely identifies each symbol with a word from the document being analyzed. No output to the user is given with the exception of a list of the results of each sentence analyzed. The processor reads input from the input tape until an end-of-sentence terminator is discovered, processes the sentence, and then writes the output tape with the results. Then the next sentence is read from the input tape and analyzed. This is repeated until the input tape is exhausted and then control returns to LETAP. A diagram of the LEADERMART Text Analysis Procedure is shown in Figure 5.3.

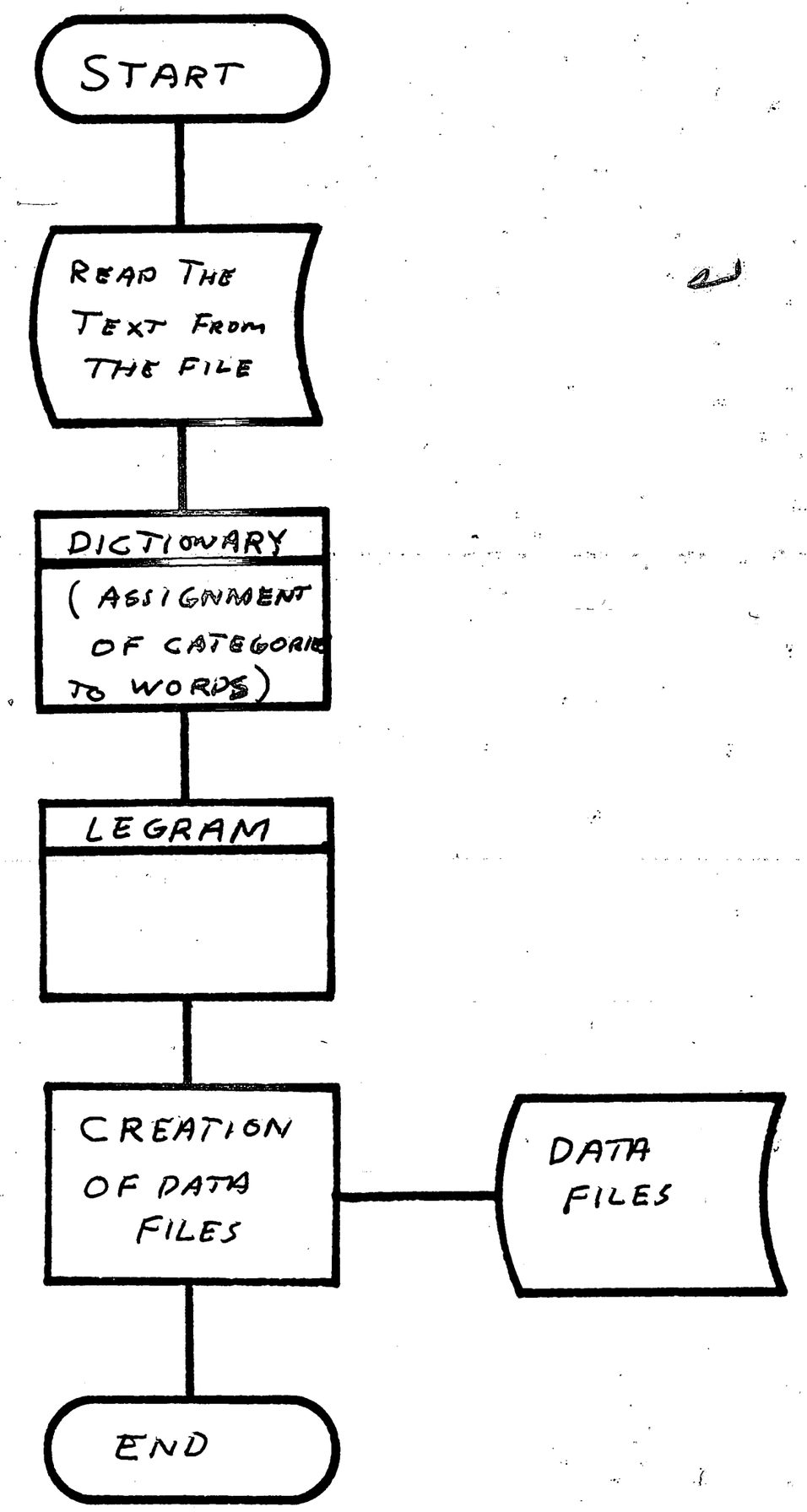


Figure 5.3. The LETAP Process

VI. Conclusion

A. Results

As a result of rewriting the SMILES LOADR program and a careful investigation into the organization of the SMILES processor, many improvements to the programming system have been made. This is evidenced by a reduction in the execution time of a typical job by 60 per cent. Just as important, the carful study of the SMILES Programming System showed several weaknesses in the organization of the programs.

A significant result was the overall improvement of the LOADR program when a formal study of the problem was made before the rewriting was initiated. The time taken to study the language, transformations, and compiling principles paid off with a program which is easy to debug, follow, and modify. In a situation where many people have access to the same program, these characteristics are ideal.

B. Future Developments of the SMILES Language

During the study of the SMILES Programming System, many programming deficiencies were noted in the SMILES execution program. One example is the search technique used in the Inclusion List utilization where a binary search would be much faster than the present method. These possible improvements have been recorded and will probably be made along with some organization changes to the internal flow of the program.

A major area of investigation will probably be the implementation of a look-ahead paging scheme in the SMILES processor. This would require an area which would hold two pages in core at one time instead of a single page as is done now. While the SMILES Processor is executing the program in one page, the disk I/O routine could be bringing the next expected page from the disk into the memory of the SMILES Processor. A statistical investigation was made into the interpage transition (transfer operations between pages) and it appears that between 80 percent and 90 percent of the interpage transfers are between sequentially assigned pages. This means that between 80 and 90 percent of the time, an accurate page-ahead operation could be performed. These statistics appear to be the result of the straight-forward programming technique used in the LEGRAM program (the SMILES program used in LEGRAM). This page-ahead operation would increase the throughput of the SMILES Processor by decreasing the page-in operation dead time (the time when the SMILES Processor is idling while waiting for the disk request to be completed).

APPENDIX I

A FORMAL DEFINITION OF THE SMILES SYNTAX

<PROGRAM> ::= <MAINPROG>|<SUBS>

<SUBS> ::= <SUBR>|<SUBS> <SUBR>

<SUBR> ::= <SUBC> <RULES> <ENDC>

<SUBC> ::= SMILES SUBROUTINE <PROGNAME>

<MAINPROG> ::= <PROGC> <EXTC> <RULES> <ENDC>

<PROGC> ::= SMILES PROGRAM <PROGNAME>

<PROGNAME> ::= <LETTER>|<PROGNAME> <LETTER>

<RULES> ::= <RULE>|<RULES> <RULE>

<RULE> ::= <TRNSLAB> <STAR> <IF> <THEN> <STAR>
<TRNSFERS> <STAR>|<TRANLAB> <STAR>
<SETOP> <STAR> <TRNSFERS> <STAR>

<EXTC> ::= EXTERNAL <PROGNAME>

<TRNSLAB> ::= <ALPHANUM>|<TRNSLAB> <ALPHANUM>

<IF> ::= <IFOPS> <EQUALS>

<IFOPS> ::= <OP>|<IFOPS> <OP>|"<" <IFOPS> ">" |
"(" <IFOPS> ")"

<OP> ::= <CATLAB>|<OPERATOR>|<REGISTER>

<CATLAB> ::= <LETTER>|<CATLAB> <ALPHANUM>

<OPERATOR> ::= $ | + |; | :: NUMBER

<REGISTER> ::= <PERIOD> <NUMBER> <PERIOD>

<SETOP> ::= SET <REGISTER> <EQUALS> <CATLAB>

<THEN> ::= <THENOPS>|<THEN> <THENOPS>

<THENOPS> ::= <NUMBER>|<SLASHOP>

<SLASHOP> ::= <SLASH1>|<SLASH2>|<SLASH3>|
<SLASH4>|<SLASH5>|<SLASH6>

<SLASH1> ::= <NUM> <SLASH> <COMMA> <NUM>

<SLASH2> ::= <NUM> <SLASH> <NUM>

<SLASH3> ::= <CATLAB> <SLASH> <NUM>

<SLASH4> ::= <CATLAB> <SLASH> <COMMA>

<SLASH5> ::=<CATLAB> <SLASH> <NUM> <COMMA> <NUM>

<SLASH6> ::= <SLASH> <NUM>

<TRANSFERS> ::= <TRNSLAB> <COMMA> <TRNSLAB>|
<COMMA> <TRNSLAB>|<TRNSLAB>

<SLASH> ::= /

<STAR> ::= *

<COMMA> ::= ,

<ALPHANUM> ::= <ALPHA> | <NUMBER>

<ALPHA> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|
S|T|U|V|W|X|Y|Z

<NUMBER> ::= 1|2|3|4|5|6|7|8|9|0

APPENDIX II

Transformations of the Loading Process

A. The "IF" Section

CATLAB = CATLAB

SET = ↑S

PERIOD NUMBER PERIOD = ↑R NUMBER

:: = ↑C

NUMBER = NUMBER

OPERATOR = OPERATOR

EQUALS SIGN = EQUALS SIGN

B. The "THEN" Section

N_1 "/" N_2 "," N_3 = "/M" N_1 N_2 N_3

N_1 "/" N_2 = "/L" N_1 N_2

SYM "/" N_1 = "/L" SYM N_1

SYM "/" "," = "/L" SYM 0 0

SYM "/" N_1 "," N_2 = "/M" SYM N_1 N_2

"/" N_1 = "/K" 0 N_1

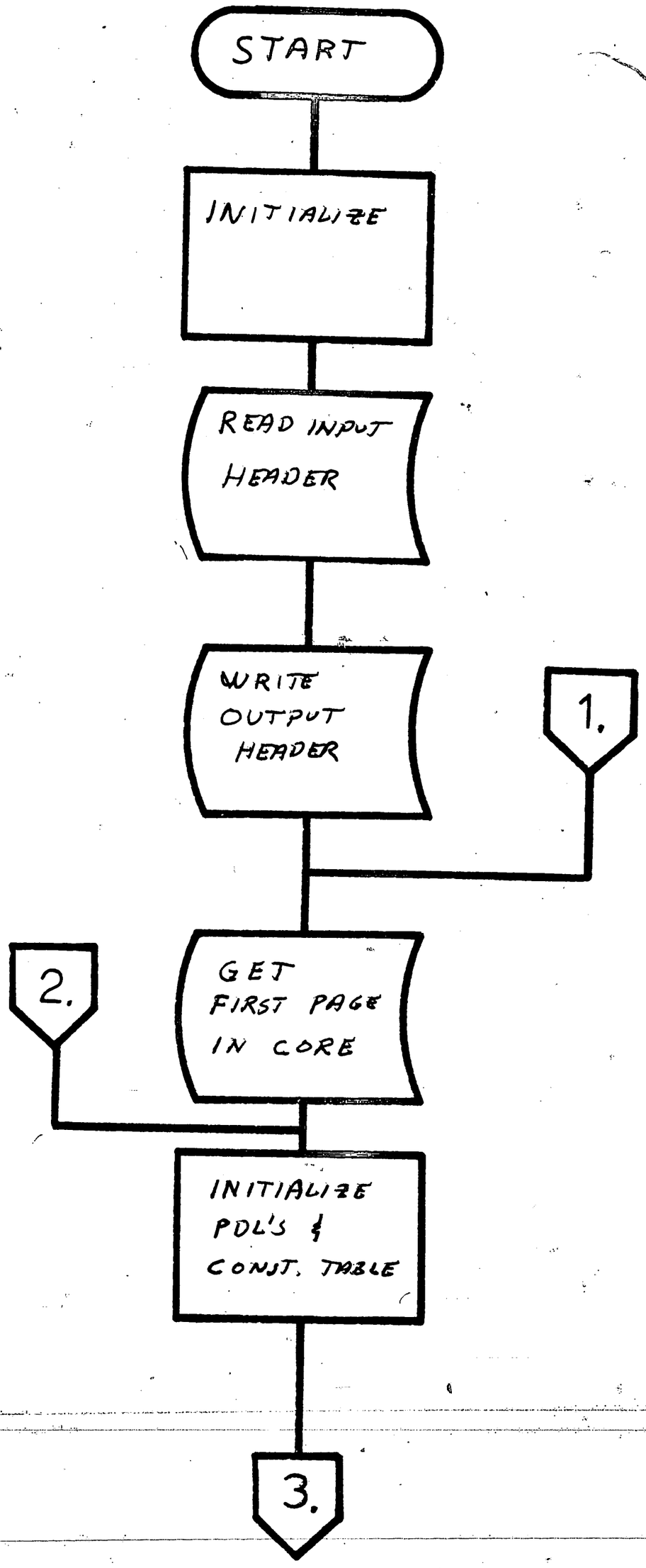
N_1 = N_1

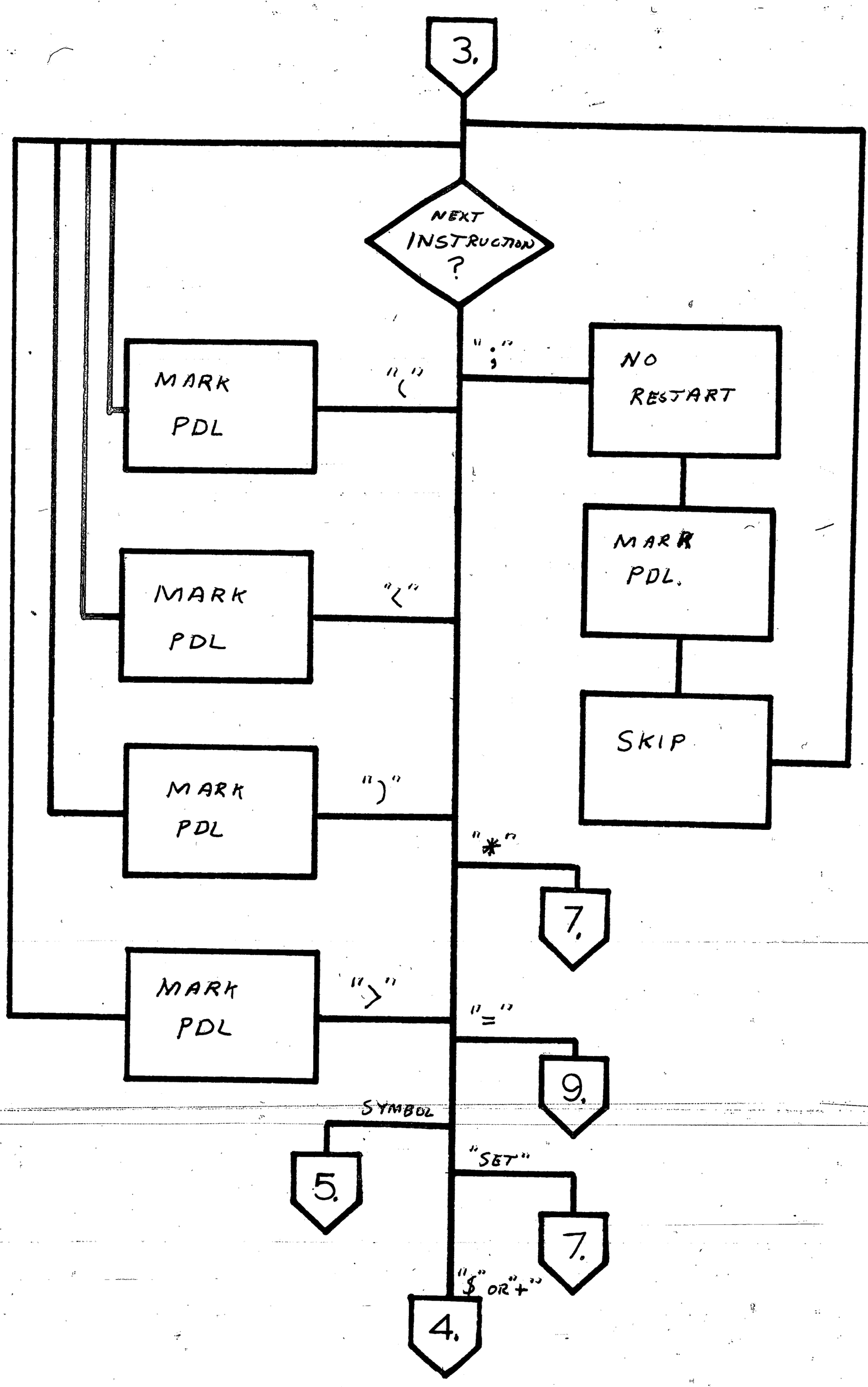
Where N_i, i=1,2,3 are numbers and

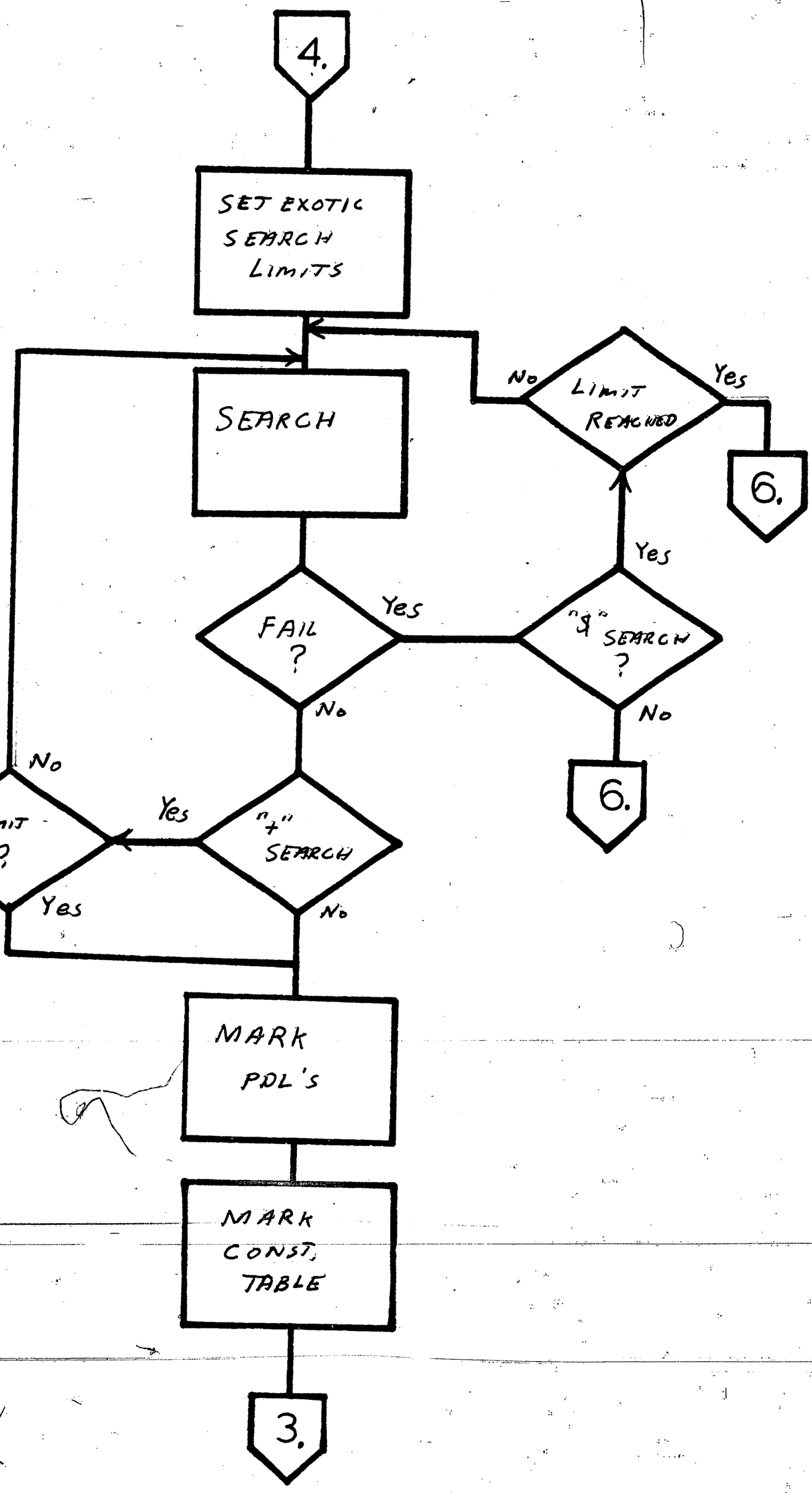
SYM are symbols

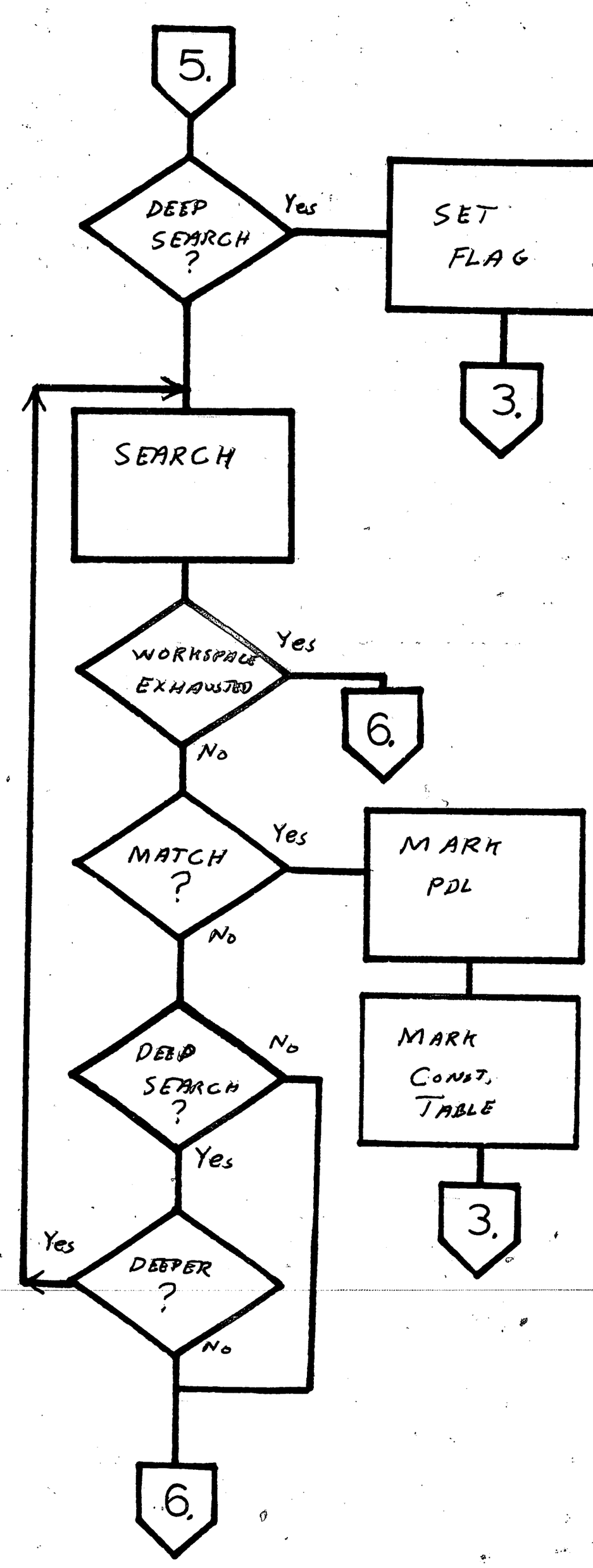
APPENDIX III

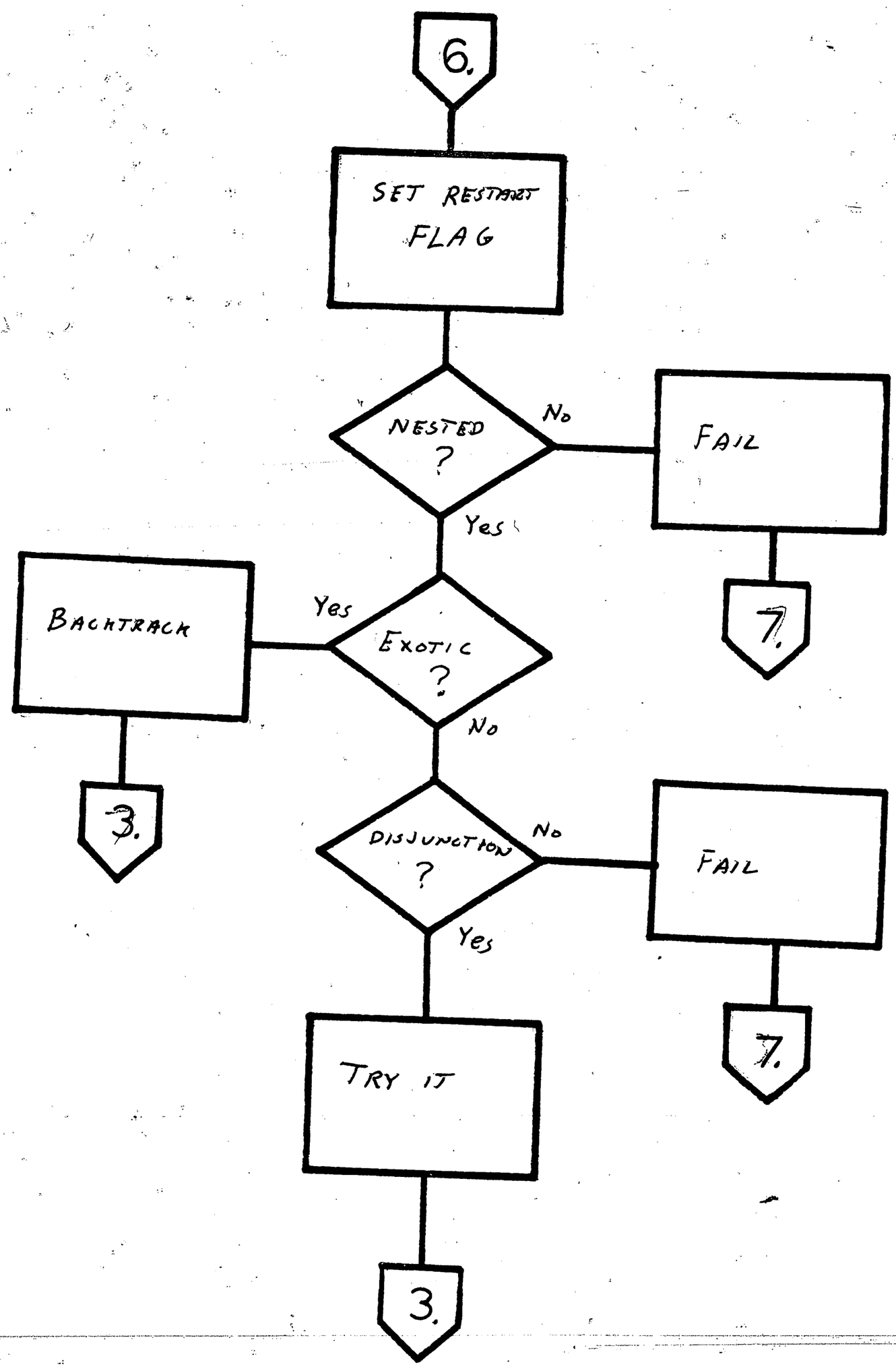
Flowchart of the SMILES Processor

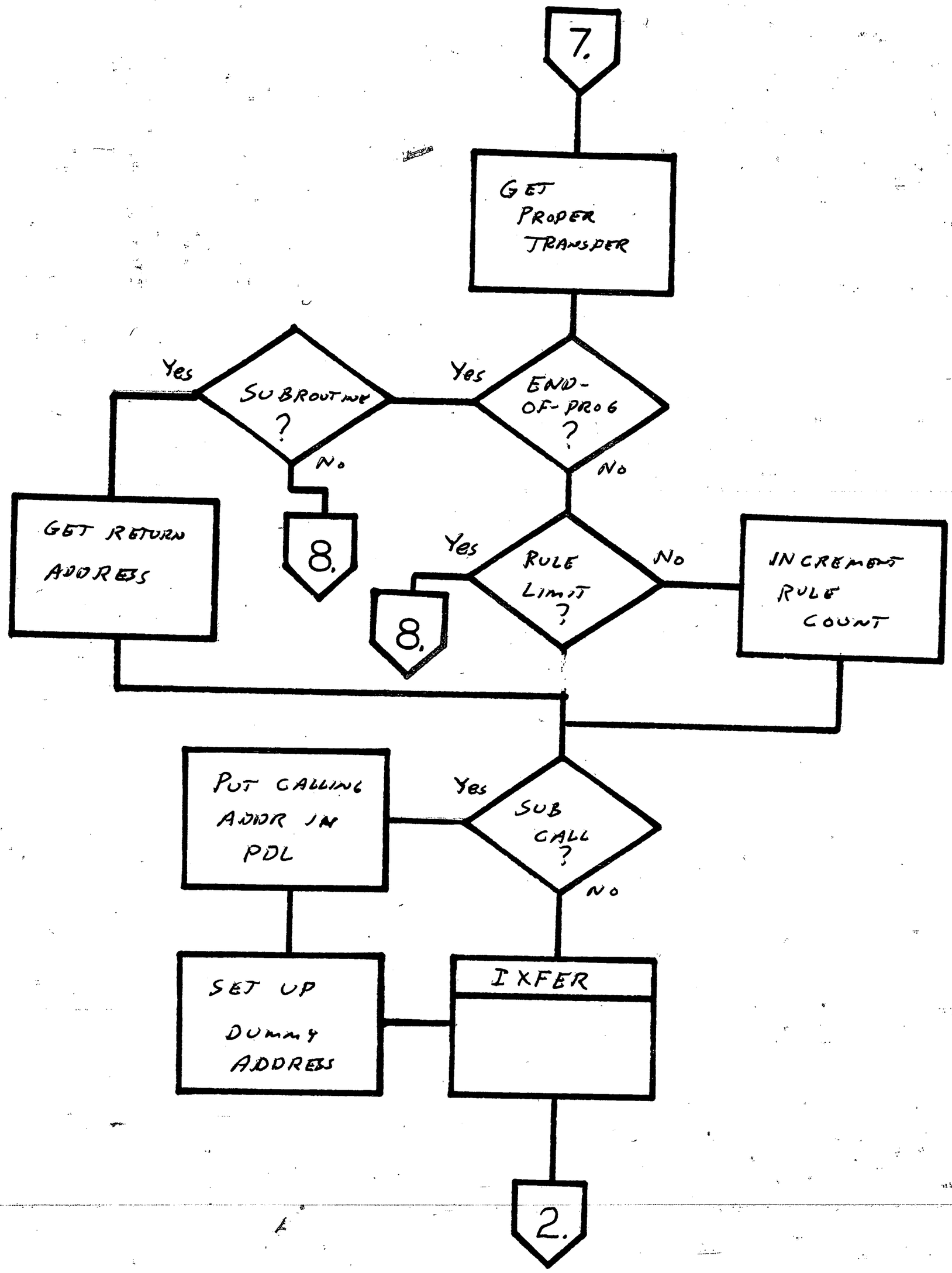


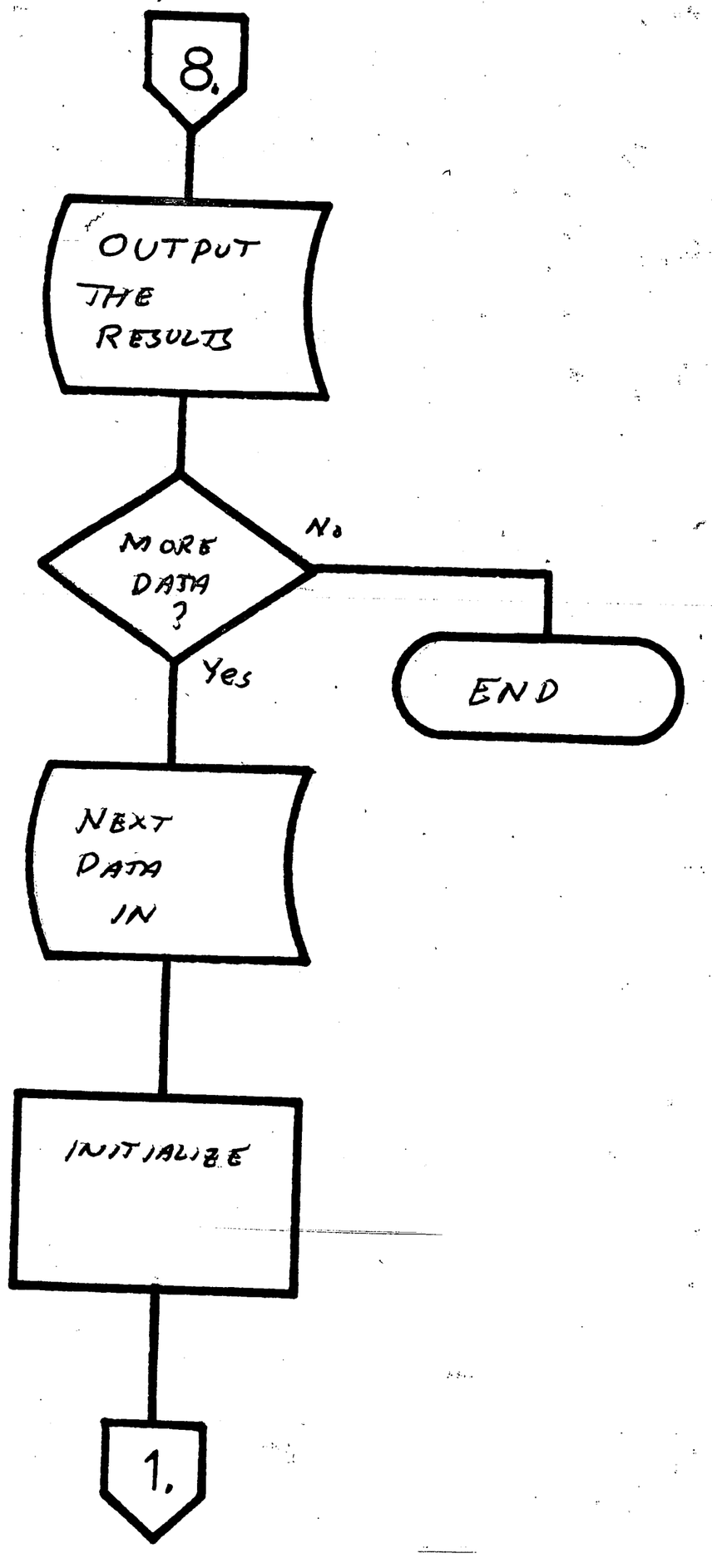

3.
NEXT INSTRUCTION ?
MARK PDL
"("
MARK PDL
"<"
MARK PDL
")"
MARK PDL
">"
";"
NO RESTART
MARK PDL.
SKIP
"*"
7.
"="
9.
SYMBOL
"SET"
7.
"$" OR "+"
4.
5.

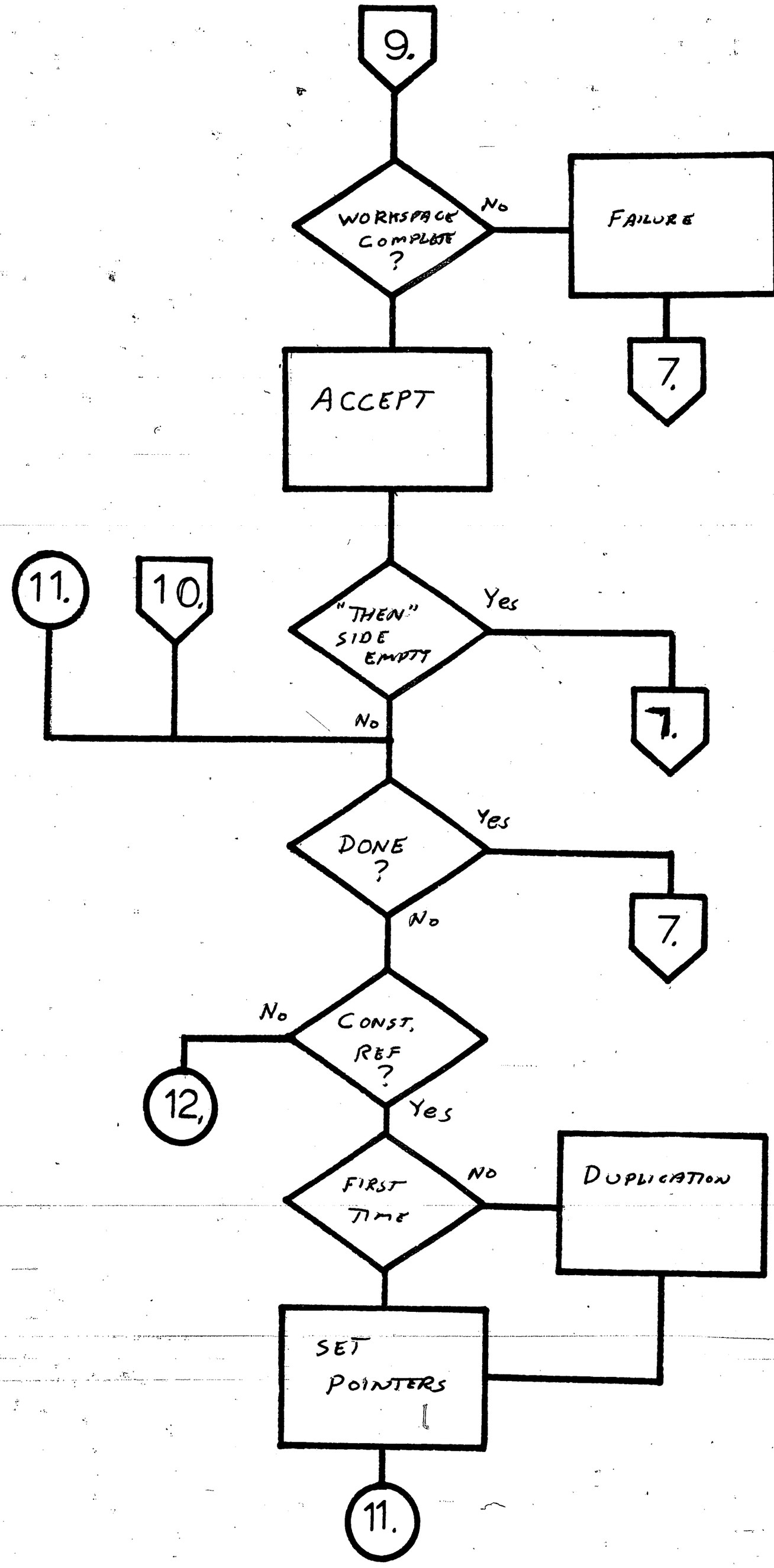

4.
SET EXOTIC SEARCH LIMITS
SEARCH
No
LIMIT REACHED
Yes
6.
Yes
FAIL ?
Yes
"3" SEARCH ?
No
No
6.
No
LIMIT ?
Yes
"+" SEARCH
Yes
No
MARK PDL's
MARK CONST. TABLE
3.

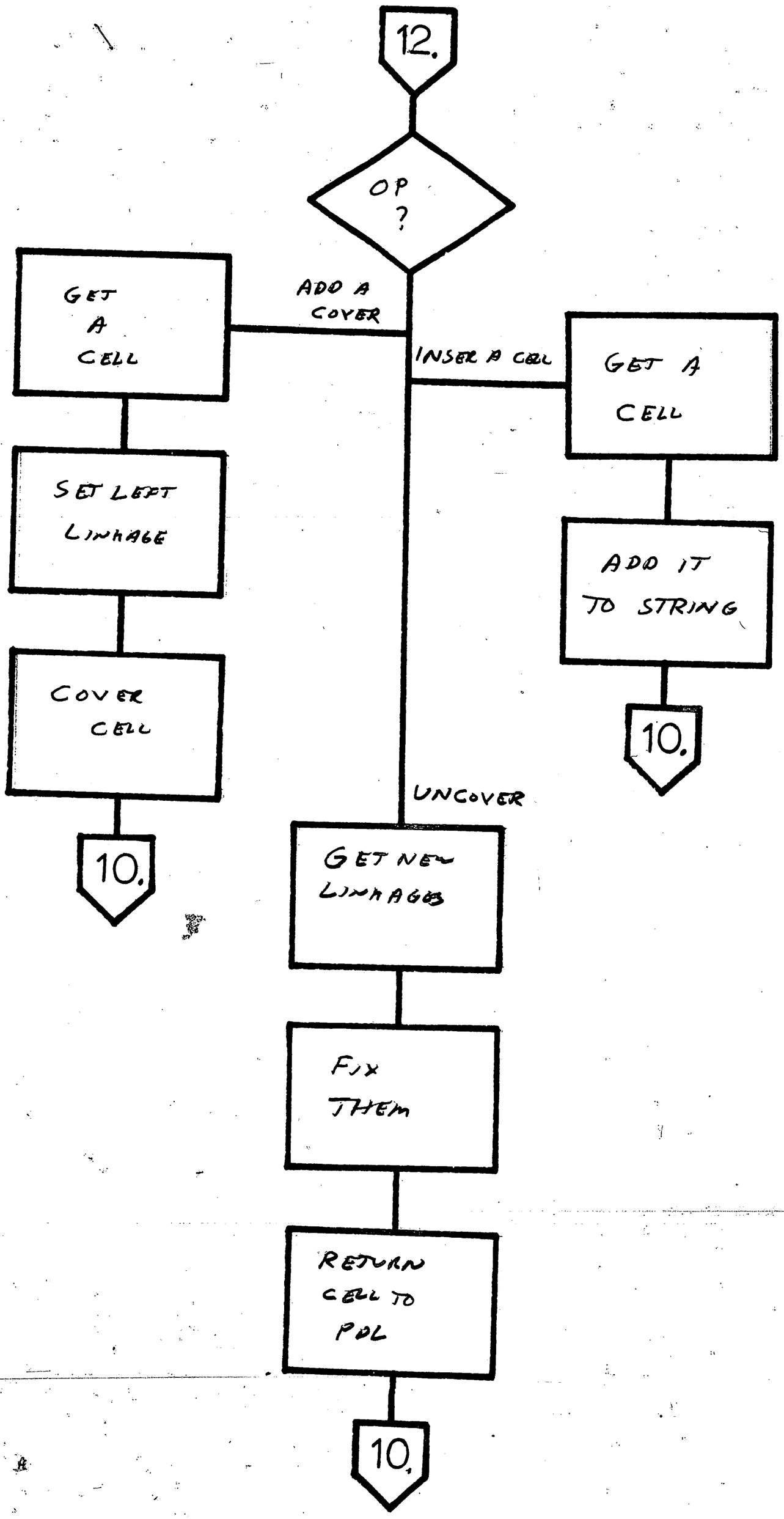

5.
DEEP SEARCH ?
Yes
SET FLAG
3.
SEARCH
WORKSPACE EXHAUSTED
Yes
6.
No
MATCH ?
Yes
MARK PDL
No
MARK CONST. TABLE
3.
DEEP SEARCH ?
No
Yes
Yes
DEEPER ?
No
6.


6.
SET RESTART FLAG
NESTED ?
No
FAIL
7.
Yes
BACKTRACK
Yes
EXOTIC ?
No
3.
DISJUNCTION ?
No
FAIL
7.
Yes
TRY IT
3.


7.
GET PROPER TRANSFER
END-OF-PROG ?
Yes
No
SUBROUTINE ?
Yes
No
8.
GET RETURN ADDRESS
8.
RULE LIMIT ?
Yes
No
INCREMENT RULE COUNT
PUT CALLING ADDR IN PDL
SUB CALL ?
Yes
No
SET UP DUMMY ADDRESS
IXFER
2.


8.
OUTPUT THE RESULTS
MORE DATA ?
No
Yes
END
NEXT DATA IN
INITIALISE
1.


9.
WORKSPACE COMPLETE ?
No
FAILURE
7.
ACCEPT
11.
10.
"THEN" SIDE EMPTY
Yes
7.
No
DONE ?
Yes
7.
No
CONST. REF ?
No
12.
Yes
FIRST TIME
NO
DUPLICATION
SET POINTERS
11.


12.
OP ?
ADD A COVER
GET A CELL
SET LEFT LINKAGE
COVER CELL
10.
INSER A CEL
GET A CELL
ADD IT TO STRING
10.
UNCOVER
GET NEW LINKAGES
FIX THEM
RETURN CELL TO POL
10.

APPENDIX IV

A Sample SMILES Execution Output

```
SMILES EXECUTION PROGRAM          CENTER FOR THE INFORMATION SCIENCES    LEHIGH UNIVERSITY


  2  TH
  3  NA
  4  NA
  5  U
  6  OF
  7  U
  8  NA
  9  BH
 10  IN
 11  ZU
 12  E9
 13  E9
 14  DP

INITIAL STRUCTURE

$
↑
TH-NA-NA-U -OF-U -NA-BH-IN-ZU-E9-E9-DP
        CP TIME AT START OF ANALYSIS IS  18.991 SECONDS
        FINAL STRUCTURE  482 RULES CONSIDERED

$
↑
S
↑
SA                                      -DP
↑
NS                  -VP    -O4
↑                    ↑      ↑
NM        -PP        SV     NN
↑          ↑         ↑      ↑
A9   -U   P -PO      VC-P   A9    -E9
↑          ↑  ↑      ↑  ↑   ↑
A -N       OF NM     BH IN  A -N
↑  ↑          ↑              ↑  ↑
A1 NA         U -N           A7 E9
↑                ↑           ↑
NA               NA          ZU

CP TIME AT CONCLUSION  26.978  SECONDS

TOTAL TIME USED WAS :  7.987  CP SECONDS        4.719  PP SECONDS
   7 SUBROUTINE CALLS
  54 CELLS USED
  30 PAGE-IN OPERATIONS REQUIRED
```

APPENDIX V

Listing of the LOADR Program

```
*          I F    S I D E    P R O C E S S O R                          LOAD2540
           SPACE  3                                                      LOAD2541
LOADR.1    BSSZ   1                                                      LOAD2542
           RJ     INITZE            INITIALIZE CONSTANT REGISTERS.      LOAD2543
           CNTRLCD                                                       LOAD2544
           OPEN   OUTPUT,WRITE,RECALL                                    LOAD2545
           OPEN   RULES,WRITE,RECALL                                     LOAD2546
           OPEN   SYMFILE,WRITE,RECALL                                   LOAD2547
           HDR    PRUNUM,RESERVE                                         LOAD2548
           NEWPROG                                                       LOAD2549
           SX6    B7                FIRST SEGMENT NUMBER                LOAD2550
           SA6    SEGNUM            STORE IT.                           LOAD2551
           SUBTTL SMLMAIN                                               LOAD2552
START      NAME                     START THE NEXT PROGRAM OR SUBR.     LOAD2553
           SB3    RULESPC           SET THE OUTPUT POINTER.             LOAD2554
NXTCARD    BSSZ   0                                                     LOAD2555
           SX7    B3                                                    LOAD2556
           SA7    LASTRUL                                               LOAD2557
           CARDIN                                                       LOAD2558
           NONBLK                   SKIP THE BLANKS                     LOAD2559
           ADVANCE                  ALIGN THE POINTERS                  LOAD2560
           LABEL                    CHECK FOR TRANSFER LABEL            LOAD2561
           NONBLK                                                       LOAD2562
           STAR                     CHECK ON FIRST STAR                 LOAD2563
           JMP    ERR05             NO-STAR **ERROR**                   LOAD2564
           ADVANCE                  ALIGN THE POINTERS                  LOAD2565
           NONBLK                   GET TO THE NEXT NON BLANK           LOAD2566
           ADVANCE                  CLEAR THE POINTERS.                 LOAD2567
           STAR                     CHECK FOR TRIPLE STAR RULE.         LOAD2568
           JMP    NEXT              NOPE, MUST BE A NORMAL RULE.        LOAD2569
           PUTOUT                   WAS A TRIPLE STAR RULE, NOW WRITE   LOAD2570
*                                   OUT THE STAR AND THEN GO TO THE     LOAD2571
*                                   TRANSFER SECTION.                   LOAD2572
           JMP    TRANS                                                 LOAD2573
NEXT       NONBLK                   GET THE NEXT NONBLANK               LOAD2574
           ADVANCE                                                      LOAD2575
           OPERA                    THIS POINTS STARTS THE PROCESSING   LOAD2576
*                                   OF THE ACTUAL *IF SIDE*, AND THE    LOAD2577
*                                   PROGRAM BRANCHES BACK TO THIS POINT LOAD2578
*                                   AFTER IT IS FINISHED PROCESSING THE LOAD2579
*                                   SYMBOL,OPERATOR, ETC. IT IS         LOAD2580
*                                   PROCESSING.                         LOAD2581
           SPACE  1                                                     LOAD2582
           JMP    NOP               NOT AN OPERATOR                     LOAD2583
           PUTOUT                   PUT IT IN THE OUTPUT LIST           LOAD2584
           JMP    NEXT              FIND THE NEXT ONE                   LOAD2585
NOP        SETR                     CHECK FOR REGISTER SET COMMAND.     LOAD2586
```

```
                JMP     NOSET           DID NOT FIND IT.                        LOAD2587
                PUTIT   ↑S              INDICATE REGISTER SET COMMAND.          LOAD2588
                NONBLK                  SKIP THE BLANKS                         LOAD2589
                ADVANCE                 ALIGN THE POINTERS                      LOAD2590
                PERIOD                  MAKE SURE OF THE PERIOD                 LOAD2591
                JMP     ERR17           MUST HAVE A PERIOD OTHERWISE ERROR      LOAD2592
                ADVANCE                 SKIP THE PERIOD                         LOAD2593
                NUMBER                  GET THE NUMBER                          LOAD2594
                JMP     ERR17           MUST HAVE A NUMBER HERE                 LOAD2595
                PUTOUT                  WRITE THE NUMBER                        LOAD2596
                INTCHNG                 CONVERT IT TO INTEGER                   LOAD2597
                ADVANCE                 ALIGN THE POINTERS                      LOAD2598
                PERIOD                  MAKE SURE OF THE PERIOD                 LOAD2599
                JMP     ERR17           MUST HAVE A PERIOD HERE                 LOAD2600
                NONBLK                                                          LOAD2601
                ADVANCE                 ALIGN THE POINTERS                      LOAD2602
                EQUALS                  MUST HAVE AN EQUALS SIGN.               LOAD2603
                JMP     ERR17           THATS AN ERROR.                         LOAD2604
                NONBLK                                                          LOAD2605
                ADVANCE                 SKIP THE EQUALS SIGN                    LOAD2606
                SYMBOL                  SET EQUAL THIS SYMBOL.                  LOAD2607
                JMP     ERR17           THATS AN ERROR.                         LOAD2608
                PUTOUT                  PUT IT IN THE OUTPUT LIST.              LOAD2609
                ADVANCE                 ALIGN THE POINTERS                      LOAD2610
                NONBLK                                                          LOAD2611
                ADVANCE                                                         LOAD2612
                JMP     NOSLS           LOOK FOR THE FINAL STAR                 LOAD2613
NOSET           SYMBOL                  CHECK FOR SYMBOL                        LOAD2614
                JMP     NOSYM           NOT A SYMBOL.                           LOAD2615
                PUTOUT                  WRITE IT ON THE OUTPUT LIST.            LOAD2616
                JMP     NEXT            GO AFTER THE NEXT .                     LOAD2617
NOSYM           NUMBER                  CHECK FOR A NUMBER.                     LOAD2618
                JMP     NONUM           NOT A NUMBER.                           LOAD2619
                PUTOUT                  WRITE IT ON THE OUTPUT LIST.            LOAD2620
                INTCHNG                 CONVERT IT TO INTEGER                   LOAD2621
                JMP     NEXT            GO TO THE NEXT                          LOAD2622
NONUM           EQUALS                  CHECK FOR AN EQUALS SIGN.               LOAD2623
                JMP     NOEQU           NOT AN EQUALS SIGN                      LOAD2624
                PUTOUT                                                          LOAD2625
                JMP     THENSEC         GO TO THE THEN SECTION.                 LOAD2626
NOEQU           PERIOD                  CHECK FOR A REGISTER REFERENCE.         LOAD2627
                JMP     NOPER           NOT A PERIOD.                           LOAD2628
                ADVANCE                 SKIP PAST THE PERIOD.                   LOAD2629
                PUTIT   ↑R              INDICATE REGISTER REFERENCE             LOAD2630
                NUMBER                  WHICH REGISTER                          LOAD2631
                JMP     ERR18           FAULTY FORM                             LOAD2632
                PUTOUT                  WRITE IT OUT.                           LOAD2633
                INTCHNG                 CONVERT IT TO INTEGER                   LOAD2634
                MERGE                   MERGE THE TWO WORDS.                    LOAD2635
                ADVANCE                 ALIGN THE POINTERS                      LOAD2636
                PERIOD                  MUST HAVE A FINAL PERIOD                LOAD2637
                JMP     ERR18           FAULTY FORM                             LOAD2638
                ADVANCE                 SKIP PAST THE PERIOD.                   LOAD2639
```

```
                JMP     NEXT            GO AFTER THE NEXT ONE.                  LOAD2640
 NOPER          COLON                   CHECK FOR A CONSTITUENT REFERENCE.      LOAD2641
                JMP     ERR06                                                   LOAD2642
*                                       CHARACTERS IS EXAUSTED, NOW LOOK        LOAD2643
*                                       FOR AN EMPTY RULE. ( I.E. *** )         LOAD2644
                COLON                   MUST BE ANOTHER COLON.                  LOAD2645
                JMP     ERR06                                                   LOAD2646
                NONBLK                                                          LOAD2647
                ADVANCE                 SKIP PAST THE DOUBLE-COLON.             LOAD2648
                PUTIT   ↑C              INDICATE CONSTITUENT REFERENCE          LOAD2649
                NUMBER                  WHICH CONSTITUENT.                      LOAD2650
                JMP     ERR19           MUST BE A NUMBER                        LOAD2651
                PUTOUT                  WRITE IT OUT.                           LOAD2652
                INTCHNG                 CONVERT IT TO INTEGER                   LOAD2653
                MERGE                   MERGE THE TWO WORDS.                    LOAD2654
                ADVANCE                 ALIGN THE POINTERS.                     LOAD2655
                JMP     NEXT            GO AFTER THE NEXT CHARACTER.            LOAD2656
                TITLE THEN SIDE PROCESSOR                                       LOAD2657
                SPACE   1                                                       LOAD2658
*               T H E N   S E C T I O N   P R O C E S S O R                     LOAD2659
                SPACE   3                                                       LOAD2660
 THENSEC        NONBLK                  GET TO THE FIRST NON-BLANK CHARACTER    LOAD2661
                ADVANCE                 FIX THE POINTERS                        LOAD2662
                NUMBER                  CHECK FOR A NUMBER                      LOAD2663
                JMP     NONUMT          NOT A NUMBER                            LOAD2664
                MARKIT                  IT WAS MARK IT INCASE OF A SLASH        LOAD2665
                PUTOUT                  PUT IT IN THE OUTPUT LIST               LOAD2666
                INTCHNG                 CONVERT IT TO INTEGER                   LOAD2667
                ADVANCE                 CLEAR THE POINTERS                      LOAD2668
                SLASH                   CHECK FOR AN IMMEADIATE SLASH           LOAD2669
                JMP     THENSEC         NOT A SLASH OPERATION                   LOAD2670
                PUTOUT                  IT IS A SLASH OPERATION-PUT IT OUT      LOAD2671
                PERMUTE                 INTERCHANGE THE NUMBER AND THE SLASH    LOAD2672
*                                       NOW THE MARKIT POINTS TO THE SLASH      LOAD2673
                NONBLK                  GET THE NEXT NON-BLANK                  LOAD2674
                ADVANCE                 CLEAR THE POINTERS                      LOAD2675
 SLASH1         NUMBER                  CHECK FOR A NUMBER                      LOAD2676
                JMP     ERR08           MUST BE A NUMBER                        LOAD2677
                PUTOUT                  WRITE IT OUT                            LOAD2678
                INTCHNG                 CONVERT IT TO INTEGER                   LOAD2679
                NONBLK                  NEXT NON-BLANK                          LOAD2680
                ADVANCE                 CLEAR THE POINTERS                      LOAD2681
                COMMA                   CHECK FOR A COVER OPERATION             LOAD2682
                JMP     NOCOV           NOT A COVER OPERATION                   LOAD2683
                NONBLK                  NEXT NON-BLANK                          LOAD2684
                ADVANCE                 CLEAR THE POINTERS                      LOAD2685
                NUMBER                  GET THE NEXT NUMBER                     LOAD2686
                JMP     ERR08           MUST HAVE A NUMBER HERE                 LOAD2687
                PUTOUT                  PUT IT IN THE OUTPUT LIST               LOAD2688
                INTCHNG                 CONVERT IT TO INTEGER                   LOAD2689
                CHNGIT /M               MAKE THE SLASH INTO A SLASH M           LOAD2690
*                                       TO INDICATE COVER OPERATION             LOAD2691
```

```
                JMP     THENSEC                                         LOAD2692
NOCOV           CHNGIT  /L      NOT A COVER SO /L INSTEAD OF /M         LOAD2693
                JMP     THENSEC  LOOP TO THE NEXT VICTIM                LOAD2694
NONUMT          SYMBOL          CHECK FOR A SYMBOL                      LOAD2695
                JMP     NOSYMT  GUESS NOT-                              LOAD2696
                MARKIT          MARK IT INCASE OF A SHASH OP.           LOAD2697
                PUTOUT          PUT IT IN THE OUTPUT LIST               LOAD2698
                NONBLK          NEXT NON-BLANK                          LOAD2699
                ADVANCE         CLEAR THE POINTERS                      LOAD2700
                SLASH           CHECK FOR A SYM SLASH OP.               LOAD2701
                JMP     THENSEC GUESS NOT- TO LOOP BACK                 LOAD2702
                PUTOUT          IT WAS-NOW CHECK FOR  A SPECIAL CASE    LOAD2703
                PERMUTE         FIRST INTERCHANGE THE SYMBOL            LOAD2704
*                               AND THE SLASH-NOW THE MARKIT POINTS     LOAD2705
*                               TO THE SLASH                            LOAD2706
                NONBLK          NEXT NON-BLANK                          LOAD2707
                ADVANCE         CLEAR THE POINTERS                      LOAD2708
                COMMA           THIS IS THE SPECIAL CASE                LOAD2709
                JMP     SLASH1  NOPE-NOT THE SPECIAL CASE               LOAD2710
                CHNGIT  /M      WAS THE SPECIAL CASE                    LOAD2711
                PUTIT   0       ZERO FILL THE NEXT TWO WORDS            LOAD2712
                INTCHNG         CONVERT IT TO INTEGER                   LOAD2713
                PUTIT   0                                               LOAD2714
                INTCHNG         CONVERT IT TO INTEGER                   LOAD2715
                JMP     THENSEC LOOP BACK AND TRY AGAIN                 LOAD2716
NOSYMT          SLASH           CHECK FOR AN UNCOVER OP.                LOAD2717
                JMP     NOSLS   GUESS NOT.                              LOAD2718
                PUTIT   /K      /K IS FOR AN UNCOVER                    LOAD2719
                NONBLK          GET THE NEXT NON-BLANK                  LOAD2720
                ADVANCE         CLEAR THE POINTERS                      LOAD2721
                NUMBER          GET THE CONSTITUENT NUMBER              LOAD2722
                JMP     ERR08   WHOOPS..FOUND AN ERROR.                 LOAD2723
                PUTIT   0       MUST BE PROCEDED BY A ZERO              LOAD2724
                INTCHNG         CONVERT IT TO INTEGER                   LOAD2725
                PUTOUT          NOW PUT THE NUMBER OUT                  LOAD2726
                INTCHNG         CONVERT IT TO INTEGER                   LOAD2727
                JMP     THENSEC LOOP BACK FOR THE NEXT TRY              LOAD2728
NOSLS           STAR            CHECK FOR THE END OF THE THEN SECTION   LOAD2729
                JMP     ERR20   MUST BE AN ILLEGAL CHARACTER            LOAD2730
                PUTOUT                                                  LOAD2731
                TITLE   TRANSFER SECTION PROCESSOR                      LOAD2732
                SPACE   1                                               LOAD2733
*               T R A N S F E R   S E C T I O N   P R O C E S S O R     LOAD2734
                SPACE   3                                               LOAD2735
TRANS           NONBLK          GET TO NEXT NON-BLANK                   LOAD2736
                ADVANCE                                                 LOAD2737
                TRNSLBL         CHECK FOR A TRANSFER LABEL              LOAD2738
                JMP     NOACCPT NOT A LABEL                             LOAD2739
                PUTOUT                                                  LOAD2740
                REFSYM          PLACE IT IN THE SYMBOL TABLE            LOAD2741
                JMP     REJECT  GO TO LOOK FOR REJECT TRANSFER          LOAD2742
NOACCPT         PUTIT   0       PLACE A ZERO FOR THE TRANSFER LABEL     LOAD2743
                INTCHNG         CONVERT IT TO INTEGER                   LOAD2744
```

```
REJECT   NONBLK                          GET TO THE NEXT NON-BLANK                LOAD2745
         ADVANCE                         CLEAR THE POINTERS                       LOAD2746
         COMMA                           CHECK FOR A COMMA                        LOAD2747
         JMP     NOREJCT                 NO REJECT TRANSFER                       LOAD2748
         NONBLK                          CHECK FOR THE NEXT NON-BLANK             LOAD2749
         ADVANCE                         CLEAR THE POINTERS                       LOAD2750
         TRNSLBL                         CHECK FOR A TRANSFER LABEL               LOAD2751
         JMP     NOREJCT                                                          LOAD2752
         PUTOUT                                                                   LOAD2753
         REFSYM                          PLACE IT IN THE SYMBOL TABLE             LOAD2754
         JMP     ENDTRNS                                                          LOAD2755
NOREJCT  PUTIT   0                                                                LOAD2756
         INTCHNG                         CONVERT IT TO INTEGER                    LOAD2757
ENDTRNS  NONBLK                          GET TO THE FIRST NON-BLANK               LOAD2758
         ADVANCE                         CLEAR THE POINTERS                       LOAD2759
         STAR                            CHECK FOR THE FINAL STAR                 LOAD2760
         JMP     ERR07                   MUST HAVE A STAR HERE                    LOAD2761
         JMP     NXTCARD                 THE RULE IS FINISHED-GO TO THE           LOAD2762
                                         NEXT RULE.                               LOAD2763
TITLE    END OF PROGRAM PROCESSING                                                LOAD2764
SPACE    1                                                                        LOAD2765
```

APPENDIX VI.

A Sample Inclusion List

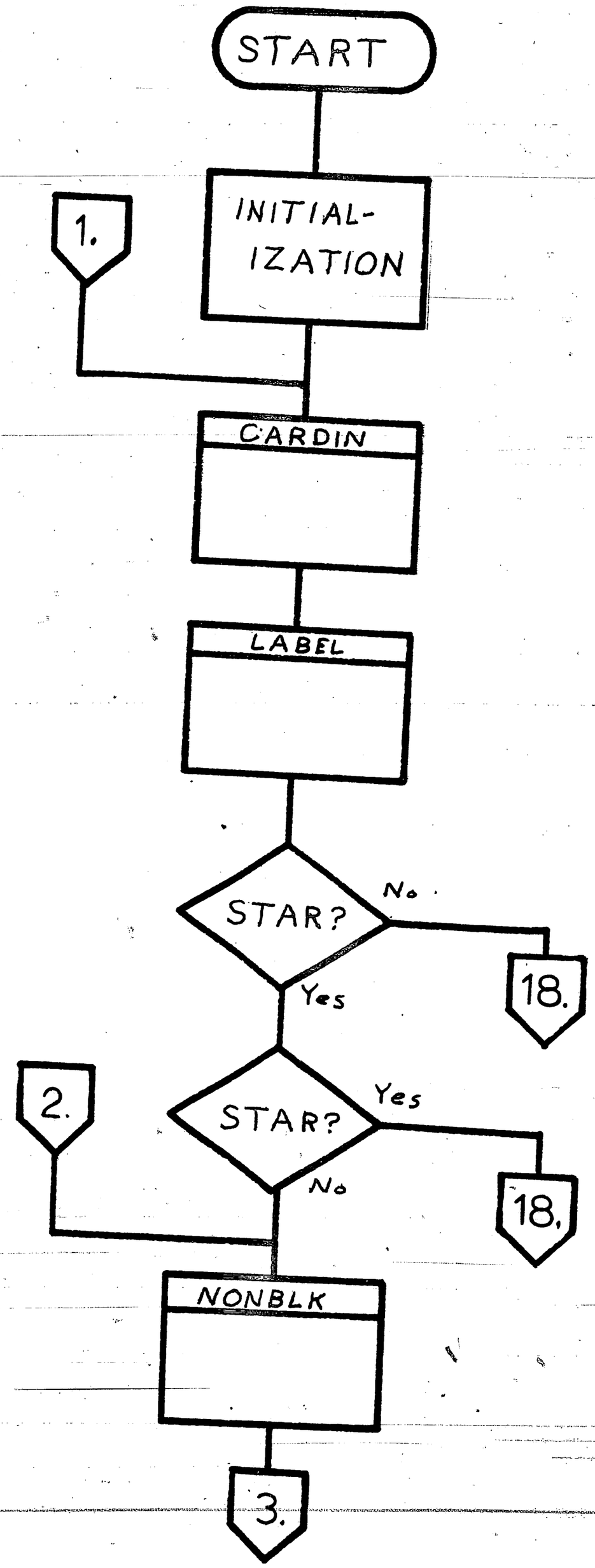
INCLUSION LIST

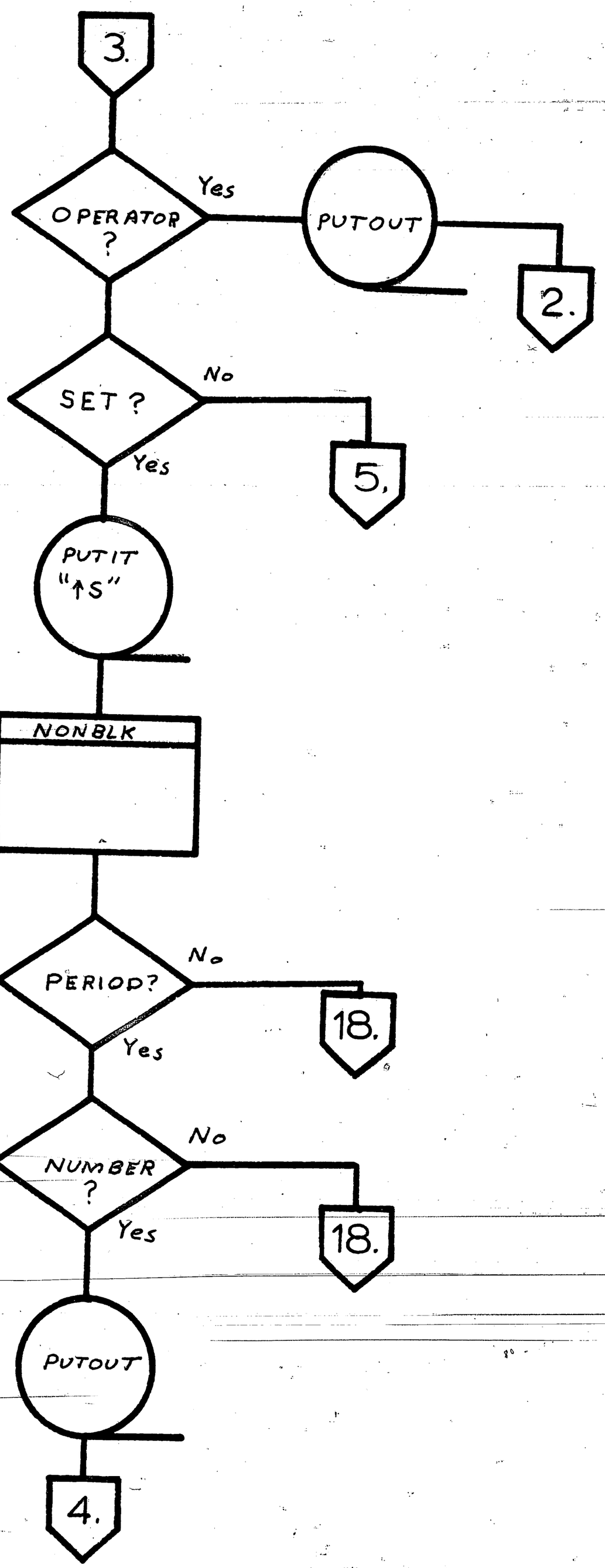
A			
	1	AA	A
	2	AB	A
	3	A1	AA
	4	A2	AA
	5	A3	AB
H			
	6	H2	HH
	7	H4	HG
J			
	8	J1	JJ
	9	J2	JJ
	10	J3	JJ
	11	J4	JJ
M			
	12	M1	M2
	13	M2	M3
	14	M3	M4
	15	M4	M5
	16	M5	M6
	17	M6	M7
	18	M7	M8
	19	M8	M9
Z			
	20	Z5	ZZ

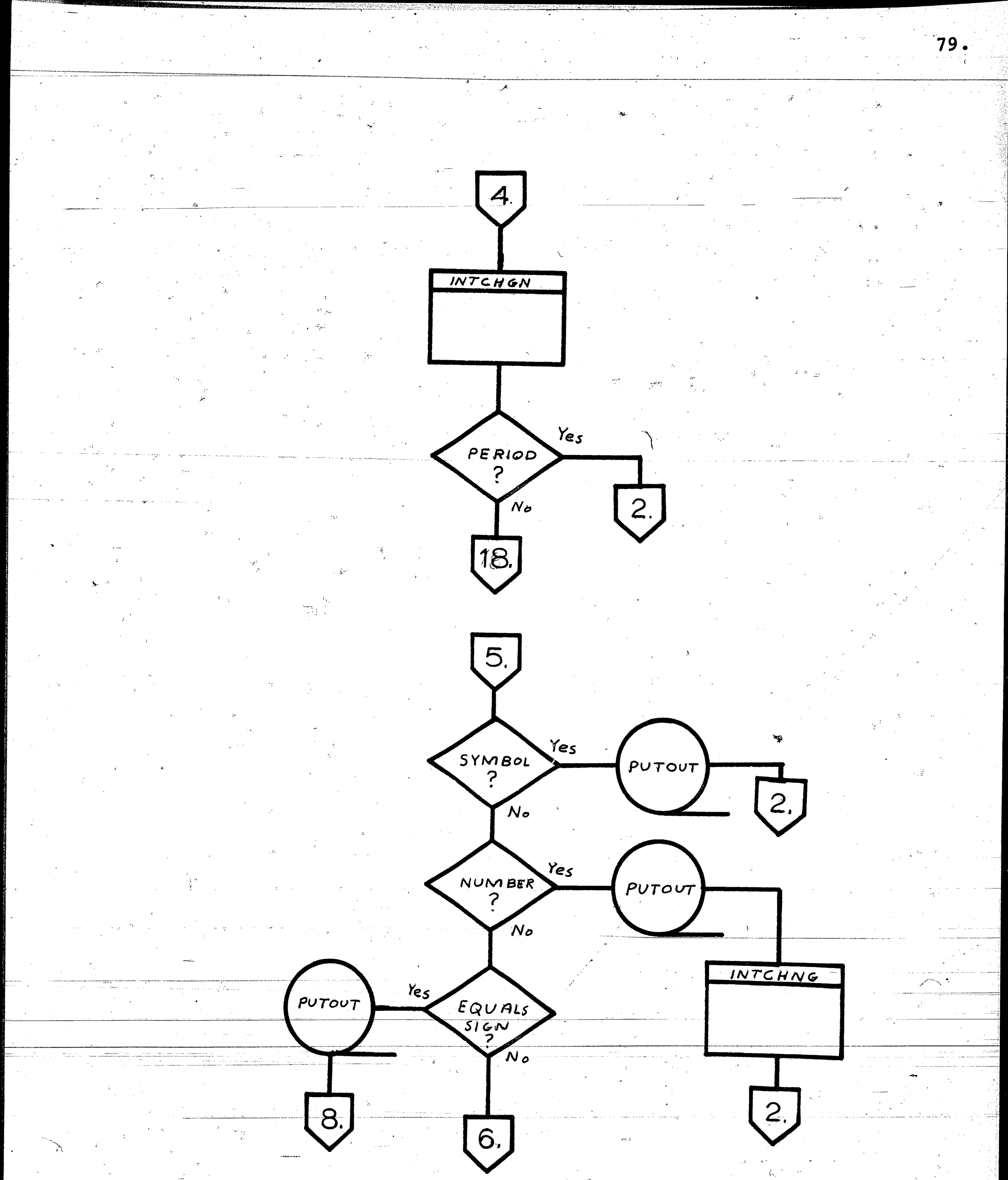
START OF THE COMBINED INCLUSION LISTING

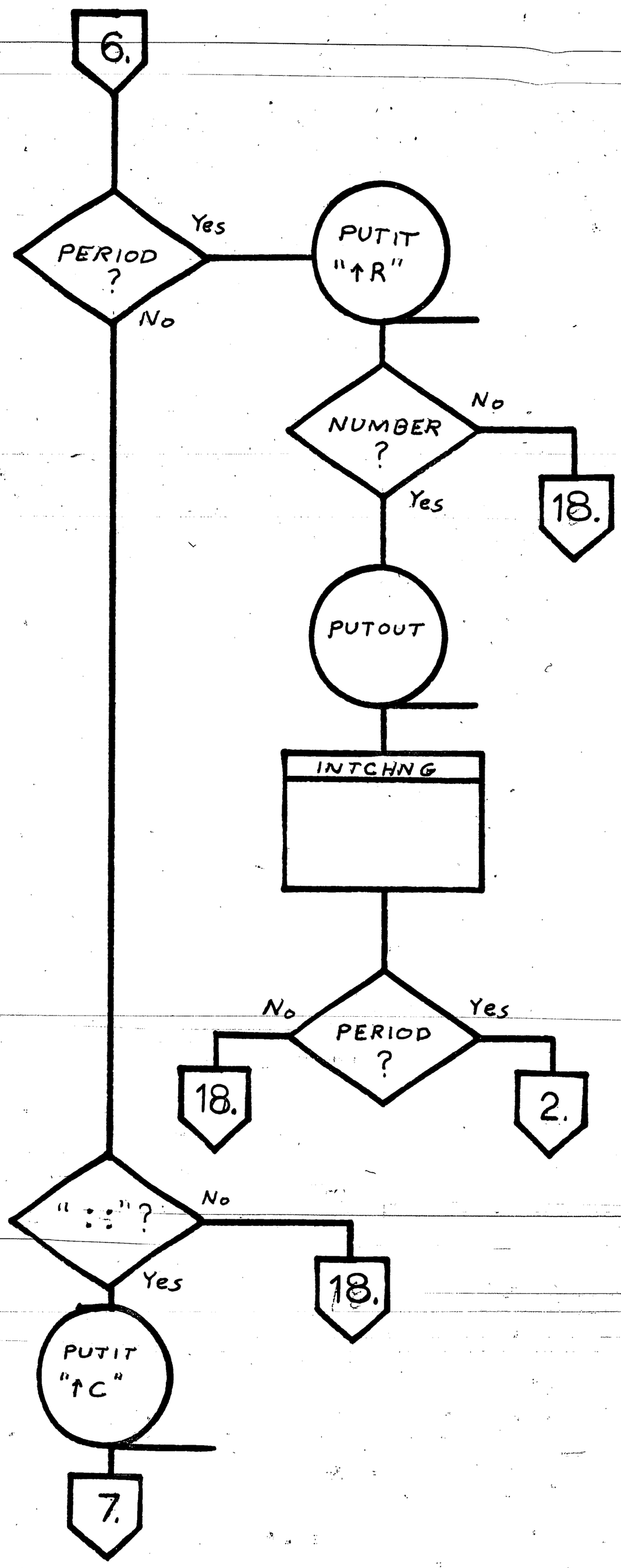
1	AA	A							
2	AB	A							
3	A1	AA	A						
4	A2	AA	A						
5	A3	AB	A						
6	H2	HH							
7	H4	HG							
8	J1	JJ							
9	J2	JJ							
10	J3	JJ							
11	J4	JJ							
12	M1	M2	M3	M4	M5	M6	M7	M8	M9
13	M2	M3	M4	M5	M6	M7	M8	M9	
14	M3	M4	M5	M6	M7	M8	M9		
15	M4	M5	M6	M7	M8	M9			
16	M5	M6	M7	M8	M9				
17	M6	M7	M8	M9					
18	M7	M8	M9						
19	M8	M9							
20	Z5	ZZ							

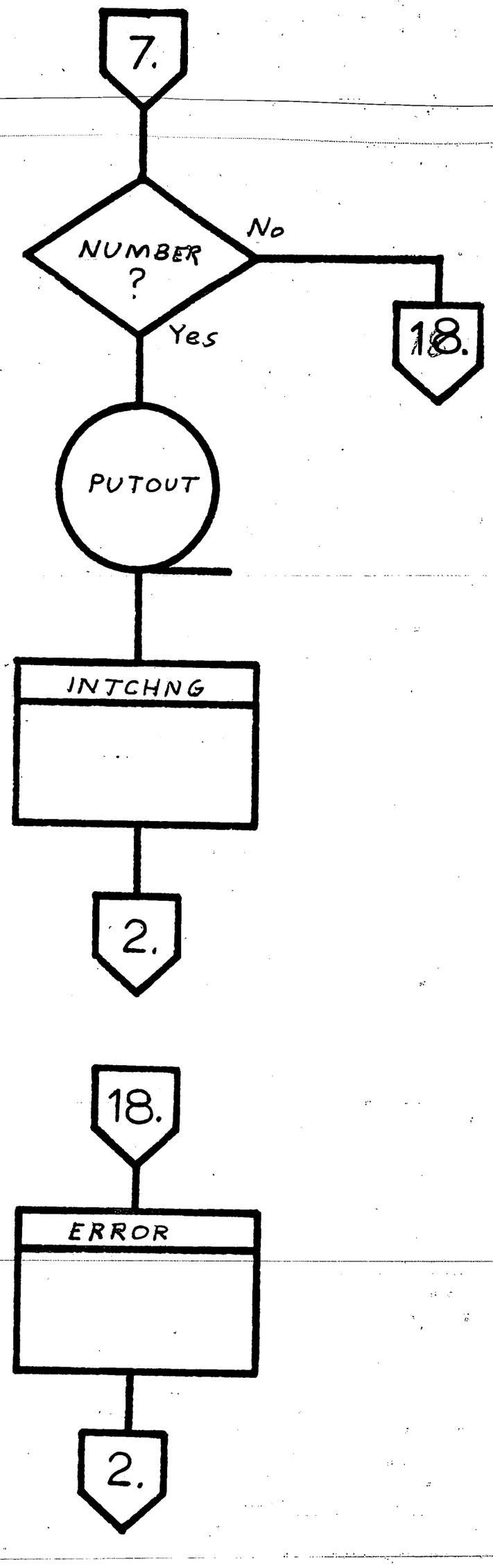
APPENDIX VII.
Flowchart of the Loading Transformations

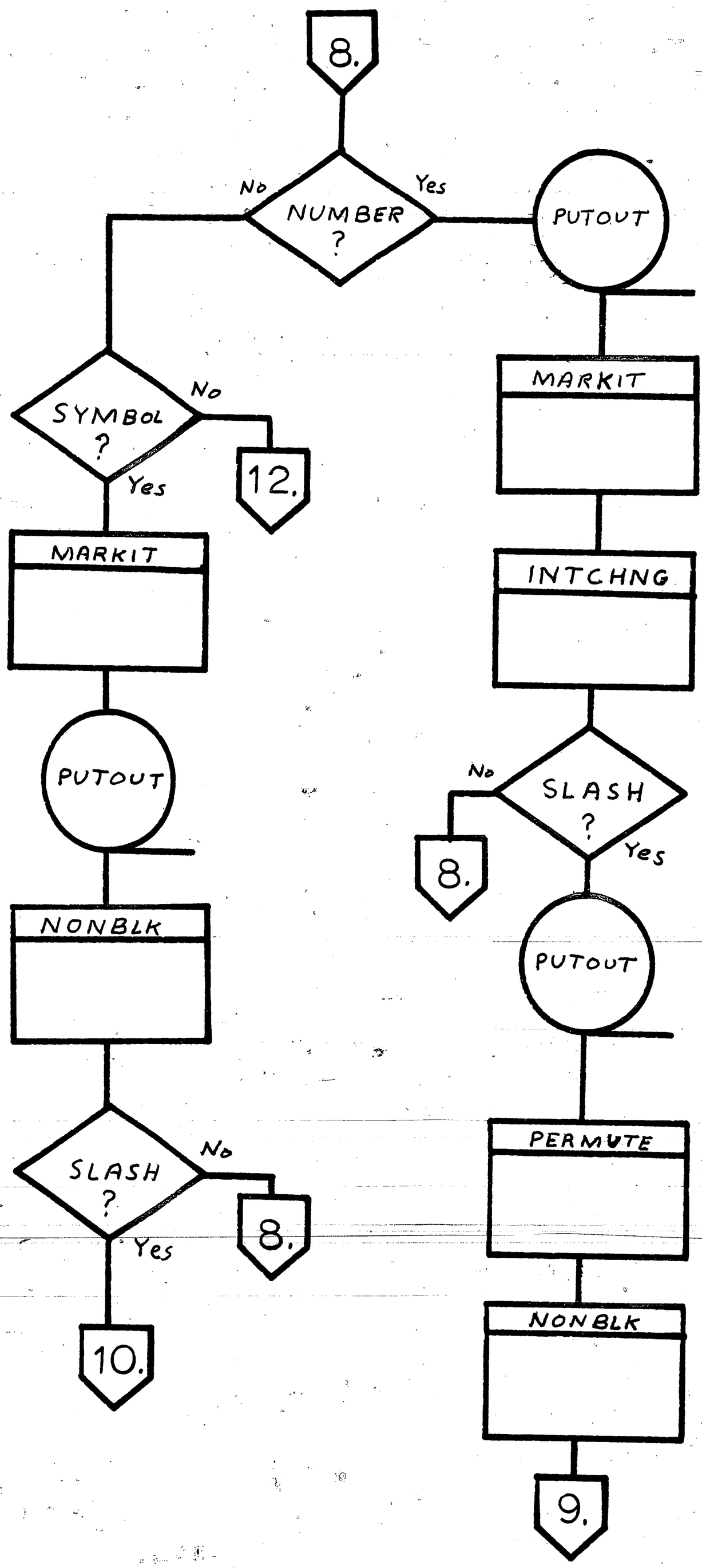


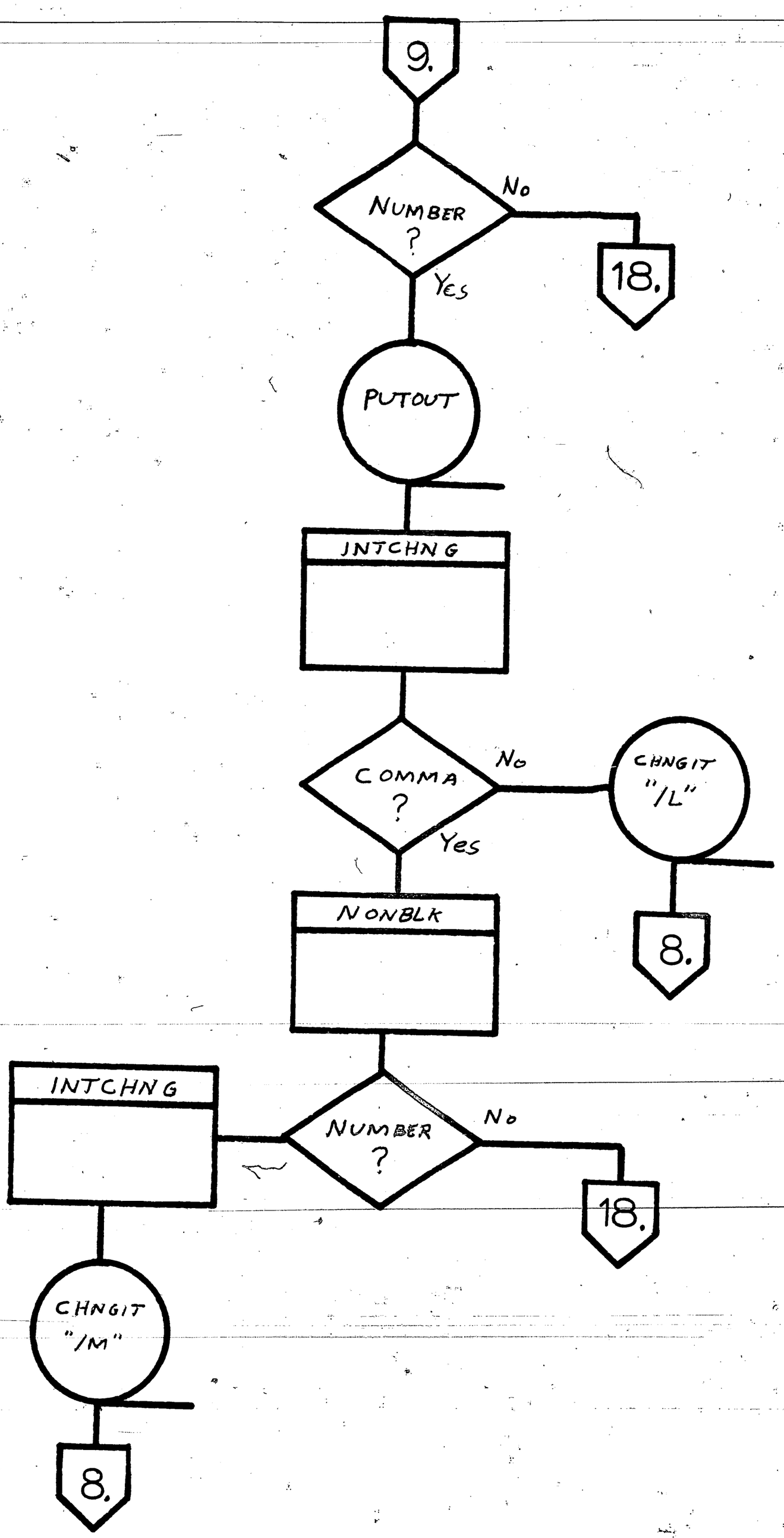

3.
OPERATOR ?
Yes
PUTOUT
2.
SET ?
No
5.
Yes
PUT IT "↑S"
NONBLK
PERIOD?
No
18.
Yes
NUMBER ?
No
18.
Yes
PUTOUT
4.

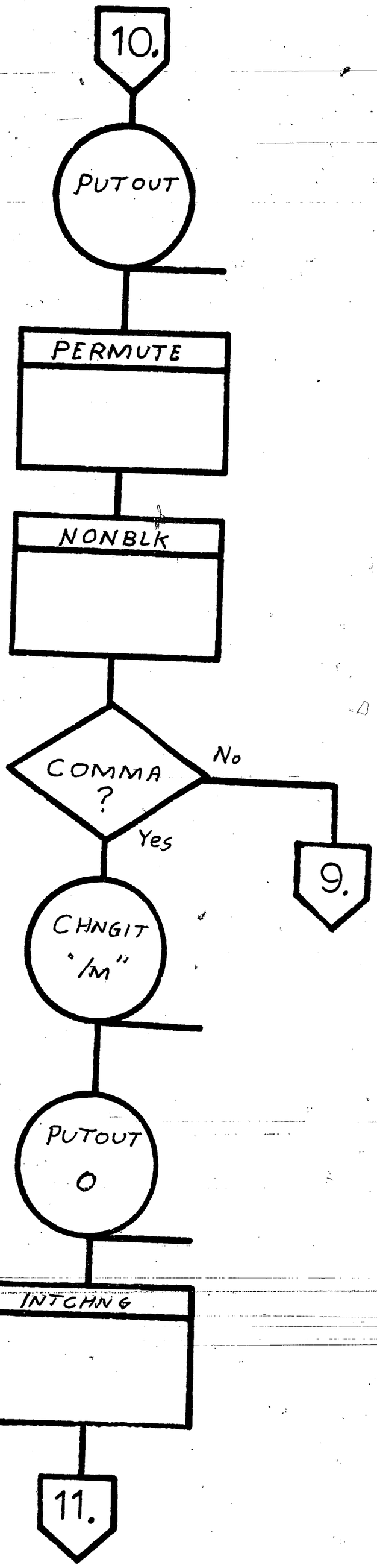

4.
INTCHGN
PERIOD ?
Yes
No
2.
18.
5.
SYMBOL ?
Yes
No
PUTOUT
2.
NUMBER ?
Yes
No
PUTOUT
INTCHNG
2.
EQUALS SIGN ?
Yes
No
PUTOUT
8.
6.

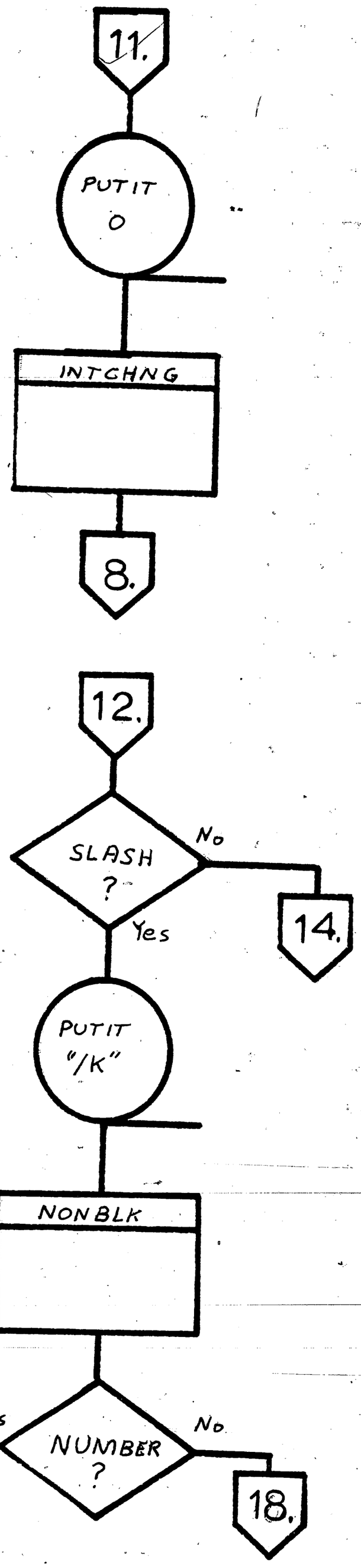

6.
PERIOD ?
Yes
PUTIT "↑R"
No
NUMBER ?
No
18.
Yes
PUTOUT
INTCHNG
No
PERIOD ?
Yes
18.
2.
" : " ?
No
18.
Yes
PUTIT "↑C"
7.

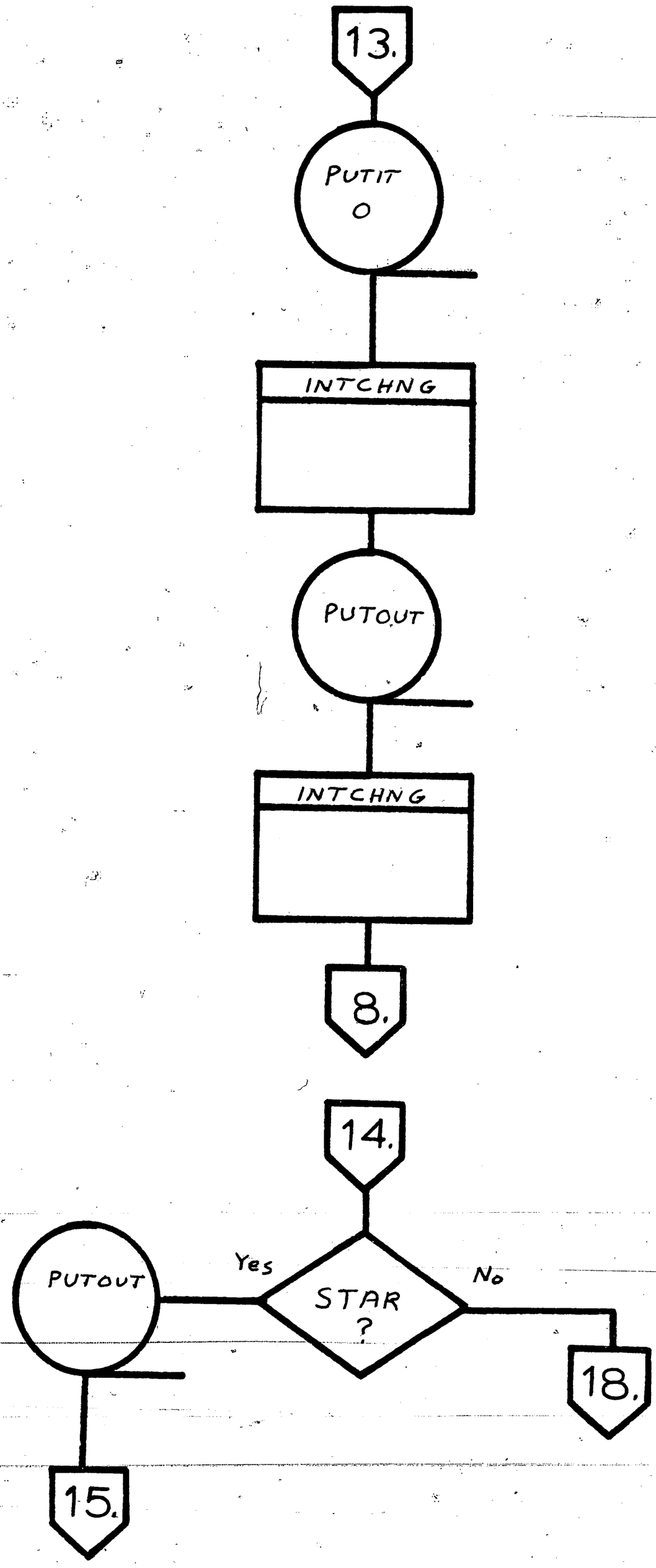

7.
NUMBER ?
No
Yes
18.
PUTOUT
INTCHNG
2.
18.
ERROR
2.

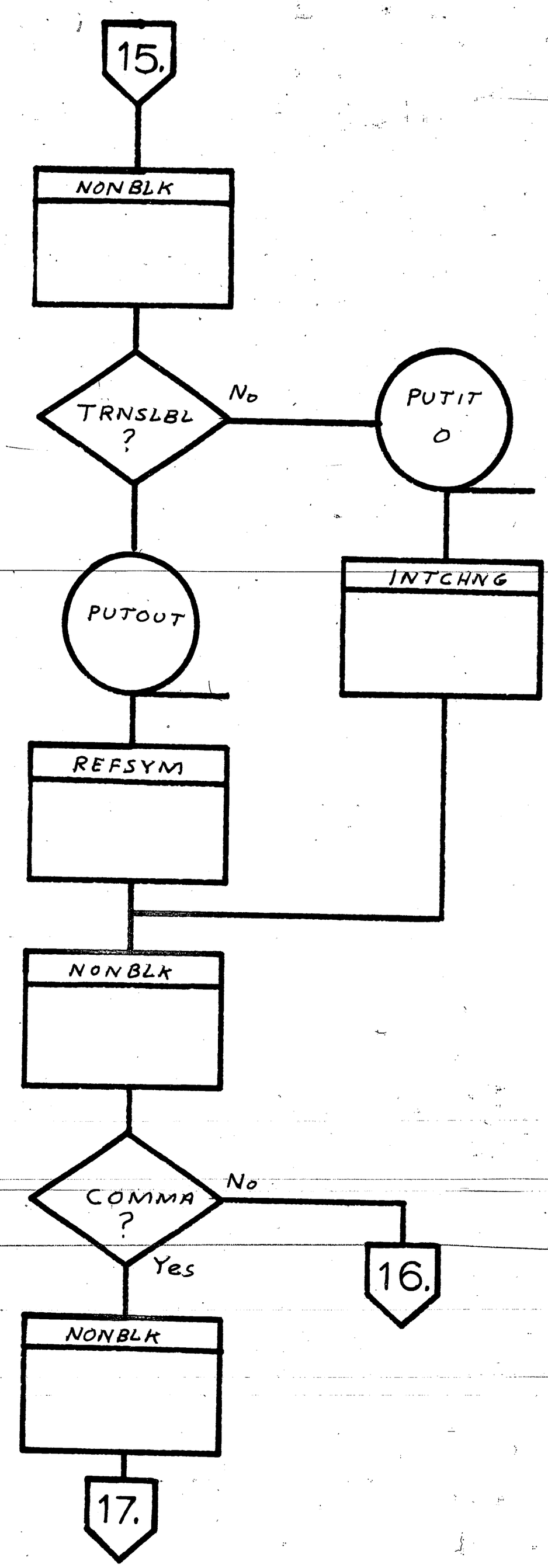

8.
NUMBER ?
No
Yes
PUTOUT
MARKIT
SYMBOL ?
No
12.
Yes
MARKIT
INTCHNG
PUTOUT
SLASH ?
No
8.
Yes
NONBLK
PUTOUT
PERMUTE
SLASH ?
No
8.
Yes
NONBLK
10.
9.

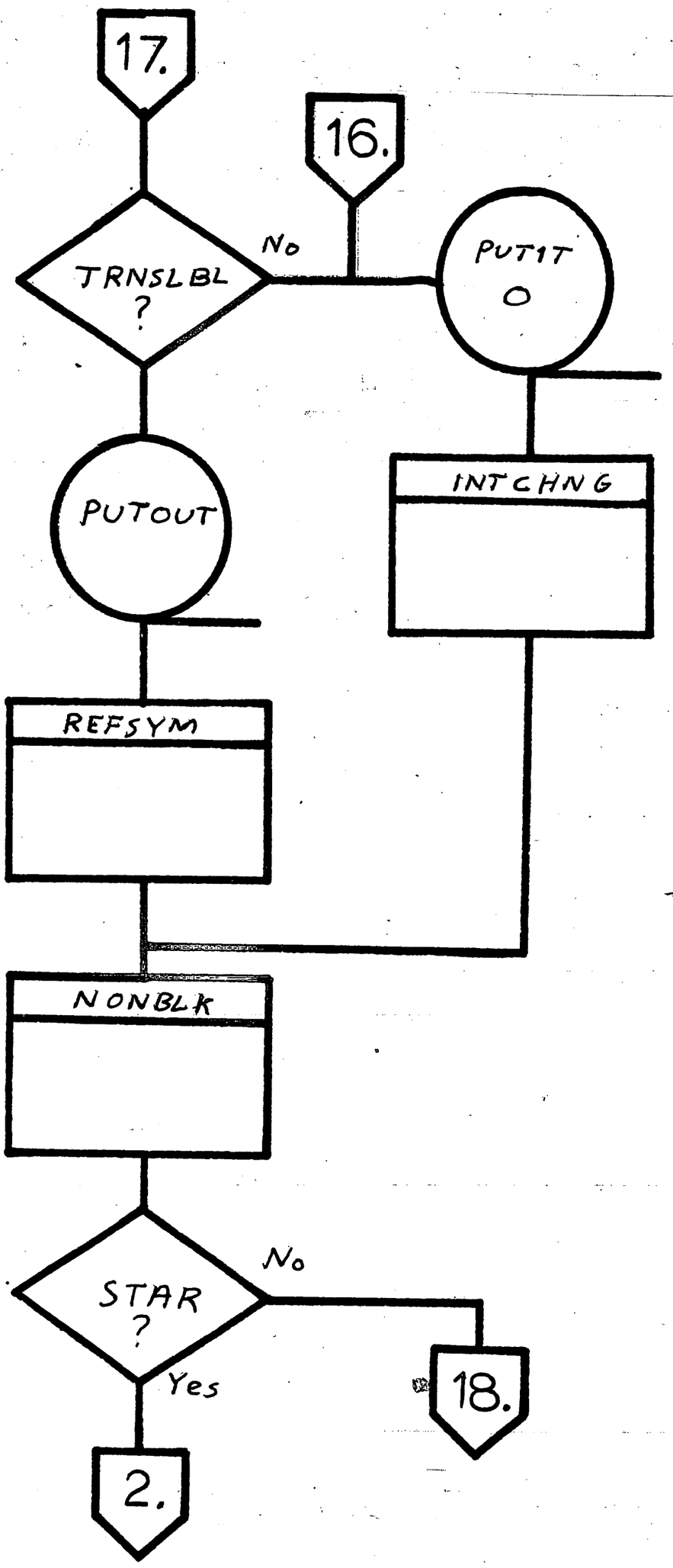

9.
NUMBER ?
No
Yes
18.
PUTOUT
JNTCHNG
COMMA ?
No
Yes
CHNGIT "/L"
8.
NONBLK
INTCHNG
NUMBER ?
No
18.
CHNGIT "/M"
8.


10.
PUTOUT
PERMUTE
NONBLK
COMMA ?
No
Yes
9.
CHNGIT "/M"
PUTOUT 0
INTCHNG
11.


11.
PUTIT
0
INTCHNG
8.
12.
SLASH
?
No
Yes
14.
PUTIT
"/K"
NONBLK
NUMBER
?
Yes
No
13.
18.


13.
PUTIT
O
INTCHNG
PUTOUT
INTCHNG
8.
14.
PUTOUT
Yes
STAR
?
No
18.
15.


15.
NONBLK
TRNSLBL ?
No
PUTIT O
PUTOUT
INTCHNG
REFSYM
NONBLK
COMMA ?
No
Yes
16.
NONBLK
17.


17.
16.
TRNSLBL
?
No
PUTIT
O
PUTOUT
INTCHNG
REFSYM
NONBLK
STAR
?
No
Yes
18.
2.

References

[1] Green, James S. "SMILES: A Symbol Manipulation Interpretive Language for Easy Syntax." Unpublished doctoral dissertation, Lehigh University, Bethlehem, Pennsylvania, June, 1970.

[2] Hillman, Donald J., and Kasarda, Andrew J. "The LEADER Retrieval System", 1969 Spring Joint Computer Conference, AFIPS Conf. Proc., Vol. 34, 1969, pp. 4470455.

[3] Hilton, W. Ralph, and Hillman, Donald J. The Structure of LECOM, Center for the Information Sciences, Lehigh University, Bethlehem, Pennsylvania, 1966.

[4] Yngve, V.H. "COMIT as an IR Language," Programming Systems and Languages, pp. 375-392, New York: McGraw-Hill, 1967.

[5] Reed, David M., and Hillman, Donald J. Document Retrieval Theory, Relevance, and the Methodology of Evaluation. Report No. 3: Microcategorization for Text-Processing, Center for the Information Sciences, Lehigh University, Bethlehem, Pennsylvania, July 7, 1966.

[6] Reed, David M. "A Computational Syntactic Analyzer." Unpublished Masters' Thesis, Information Science Department, Lehigh University, Bethlehem, Pennsylvania, 1966.

[7] Leibowitz, Michael B. "A Process for Automated Logico-Syntactic Analysis of Natural English Sentences." Unpublished Doctoral dissertation, Lehigh University, Bethlehem, Pennsylvania, 1970.

[8] Hillman, Donald J. "The Future of Information Provision." To appear in the Proceedings of the 3rd Triennial Conference of the International Association of Technological University Libraries, March 31 - April 3, 1970, Loughborough, England.

[9] Carson, John H. "The Conversion and Revision of the SMILES Language from the IBM 1800 Computer to the Control Data 6400 Computer." Private manuscript, Center for the Information Sciences, Lehigh University, Bethlehem, Pennsylvania, 1969.

[10] Amico, Antony F. Jr. Personal communications, Center for the Information Sciences, Lehigh University, Bethlehem, Pennsylvania, 1970.

[11] Wegner, Peter. Programming Languages, Information Structures, and Machine Organization. New York: McGraw-Hill, 1968.

[12] Metcalfe, Howard H. "A Parameterized Compiler based on Mechanical Linguistics," Annual Review in Automatic Programming (4), vol. 12 (1964), pp. 125-165.

[13] Barrett, William. Class notes from EE 350-s, Spring 1970, Electrical Engineering Department, Lehigh University, Bethlehem, Pennsylvania, 1970.

[14] Carson, John H. "A Multiple Output Finite State Sequential Machine." Private manuscript, Center for the Information Sciences, Lehigh University, Bethlehem, Pennsylvania, 1970.

[15] Ginsburg, Seymour. An Introduction to Mathematical Machine Theory. Massachusetts: Addison-Wesley, 1962.

[16] Control Data Corporation. SCOPE 3 Reference Manual, Palo Alto, California, Revision K, December, 1969.

[17] March, David L. Personal copy of program obtained from author, Computing Center, Lehigh University, Bethlehem, Pennsylvania, 1970.

Additional Bibliography

Kernighan, B.W., "Optimal Segmentation Points for Programs," ACM 2nd Symposium on Operating System Principles, Princeton University, October, 1969, pp. 46-53.

Lowe, Thomas C., "Automatic Segmentation of Cyclic Program Structures Based on Connectivity and Processor Timing." Communications of the ACM, vol. 13, no. 1, January, 1970, pp. 3-6, 9.

Resnick, Mark and Sable, Jerome, "ISCAM, A Syntax-Directed Language," Proceedings of the 23rd National Conference of the ACM, 1968, pp. 423-432.

Sayre, D., "Is Automatic 'Folding' of Programs Efficient Enough to Displace Manual." Communications of the ACM, vol. 12, no. 12, December, 1969, pp. 656-660.

Vita

John Hargadine Carson Jr. was born in Cleveland, Ohio, on July 21, 1947, the son of Mabel S. and John H. Carson Sr. In June, 1965, he was graduated from Rocky River High School in Rocky River, Ohio. He received a B.S. Cum Laude in Electrical Engineering in June, 1969, from Lehigh University.

He is married to the former Donna Louise Makos of Allentown, Pennsylvania, and they have no children. Mr. Carson has been employed by the Center for Information Science at Lehigh University since graduation as a research assistant aiding in the development of the LEADERMART project. He is a student member of the I.E.E.E. and the A.C.M. and has an Engineer-In-Training certification from the Pennsylvania State Registration Board for Professional Engineers.